ACTIVE DATABASE SYSTEMS

INTRODUCTION AND MOTIVATION

Traditionally, the database management system (DBMS) has been viewed as a passive repository of large quantities of data that are of interest for a particular application, which can be accessed/retrieved in an efficient manner. The typical commands for insertion of data records, deletion of existing ones, and updating of the particular attribute values for a selected set of items are executed by the DBMS upon the user's request and are specified via a proper interface or by an application program. Basic units of changes in the DBMS are the *transactions,* which correspond to executions of user programs/requests, and are perceived as a sequence of *reads* and/or *writes* to the database objects, which either *commit* successfully in their entirety or *abort*. One of the main benefits of the DBMS is the ability to optimize the processing of various queries while ensuring the consistency of the database and enabling a concurrent processing of multiple users transactions.

However, many applications that require management of large data volumes also have some behavioral aspects as part of their problem domain which, in turn, may require an ability to *react* to particular stimuli. Traditional exemplary settings, which were used as motivational scenarios for the early research works on this type of behavior in the DBMS, were focusing on monitoring and enforcing the integrity constraints in databases (1–4). Subsequently, it was recognized that this functionality is useful for a wider range of applications of DBMS. For example, a database that manages business portfolios may need to react to updates from a particular stock market to purchase or sell particular stocks (5), a database that stores users preferences/profiles may need to react to a location-update detected by some type of a sensor to deliver the right information content to a user that is in the proximity of a location of interest (e.g., deliver e-coupons when within 1 mile from a particular store) (6).

An active database system (ADBS) (1,7) extends the traditional database with the capability to *react* to various events, which can be either internal—generated by the DBMS (e.g., an insertion of a new tuple as part of a given transaction), or external—generated by an outside DBMS source (e.g., a RFID-like location sensor). Originally, the research to develop the reactive capabilities of the active databases was motivated by problems related to the maintenance of various declarative constraints (views, integrity constraints) (2,3). However, with the evolution of the DBMS technologies, novel application domains for data management, such as data streams (8), continuous queries processing (9), sensor data management, location-based services, and event notification systems (ENS) (10), have emerged, in which the efficient management of the reactive behavior is a paramount. The typical executional paradigm adopted by the ADBS is the so-called *event-condition-action* (ECA) (1,7) which describes the behavior of the form:

```
ON Event Detection
IF Condition Holds
THEN Execute Action
```

The basic tools to specify this type of behavior in commercially available DBMS are *triggers*—statements that the database automatically executes upon certain modifications. The *event* commonly specifies the occurrence of (an instance of) a phenomenon of interest. The *condition*, on the other hand, is a query posed to the database. Observe that both the detection of the event and the evaluation of the condition may require access not only to the current instance of the database but also to its history. The *action* part of the trigger specifies the activities that the DBMS needs to execute—either a (sequence of) SQL statement(s) or stored procedure calls. As a motivational example to illustrate the ECA paradigm, consider a scenario in which a particular enterprise would like to enforce the constraint that the average salary is maintained below 65K. The undesired modifications to the average salary value can occur upon: (1) an insertion of a new employee with above-average salary, (2) an update that increases the salaries of a set of employees, and (3) a deletion of employees with below-average salary. Hence, one may set up triggers that will react to these types of modifications (event) and, when necessary (condition satisfied), will perform corrective actions. In particular, let us assume that we have a relation whose schema is Employee(Name, ID, Department, JobTitle, Salary) and that, if an insertion of a new employee causes the average salary-cap to be exceeded, then the corrective action is to decrease everyone's salary by 5%. The specification of the respective trigger[1] in a typical DBMS, using syntax similar to the one proposed by the SQL-standard (11), would be:

```
CREATE TRIGGER New-Employee-Salary-Check
ON INSERT TO Employee
IF (SELECT AVG Employee.Salary) > 65,000
UPDATE Set Employee.
Salary = 0.95*Employee.Salary
```

This seemingly straightforward paradigm has generated a large body of research, both academic and industrial, which resulted in several prototype systems as well as its acceptance as a part of the SQL99 (11) standard that, in turn, has made triggers part of the commercially available DBMS. In the rest of this article, we will present some of the important aspects of the management of reactive behavior in ADBS and discuss their distinct features. In particular, in the section on formalizing and reasoning, we

[1]Observe that to fully capture the behavior described in this scenario, other triggers are needed—ones that would react to the UPDATE and DELETE of tuples in the Employee relation.

motivate the need to formalize the active database behavior. In the section on semantic dimensions, we discuss the various parameters and the impact of the choice of their possible values, as they have been identified in the literature. In the overview section we present the main features of some prototype ADBS briefly, along with the discussion of some commercially available DBMS that provide the triggers capability. Finally, in the last section, we outline some of the current research trends related to the reactive behavior in novel application domains for data management, such as workflows (12), data streams (8,9), moving objects databases (13,14), and sensor networks (6).

FORMALIZING AND REASONING ABOUT THE ACTIVE BEHAVIOR

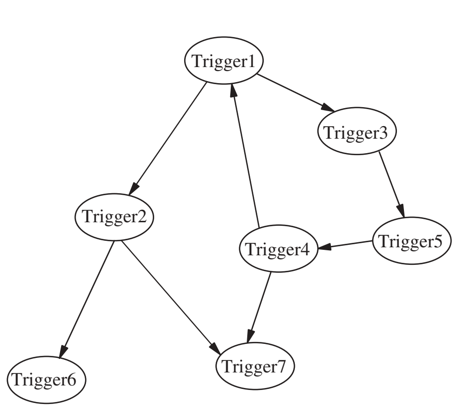
Historically, the reactive behavior expressed as a set of *condition* $\rightarrow$ *action* rules (IF *condition* holds, THEN execute *action*) was introduced in the Expert Systems literature [e.g., OPS5 (15)]. Basically, the inference engine of the system would "cycle" through the set of such rules and, whenever a left-hand side of a rule is encountered that matches the current status of the knowledge base (KB), the action of the right-hand side of that rule would be executed. From the perspective of the ECA paradigm of ADBS, this system can be viewed as one extreme point: CA rules, without an explicit event. Clearly, some kind of implicit event, along with a corresponding formalism, is needed so that the "C"-part (condition) can reflect properly and monitor/evaluate the desired behavior along the evolution of the database. Observe that, in general, the very concept of an *evolution* of the database must be defined clearly for example, the *state* of the data in a given instance together with the *activities log* (e.g., an SQL query will not change the data; however, the administrator may need to know which user queried which dataset). A particular approach to specify such conditions in database triggers, assuming that the "clock-tick" is the elementary implicit event, was presented by Sistla and Wolfson (16) and is based on temporal logic as an underlying mechanism to evaluate and to detect the satisfiability of the condition.

As another extreme, one may consider the EA type of rules, with a missing condition part. In this case, the detection of events must be empowered with the evaluation of a particular set of facts in a given state of the database [i.e., the evaluation of the "C"-part must be embedded within the detection of the events (5)]. A noteworthy observation is that even outside the context of the ADBS, the event management has spurred a large amount of research. An example is the field known as event notification systems in which various users can, in a sense, "subscribe" for notifications that, in turn, are generated by entities that have a role of "publishers"—all in distributed settings (10). Researchers have proposed various algebras to specify a set of *composite* events, based on the *operators* that are applied to the basic/primitive events (5,17). For example, the expression $E = E1 \wedge E2$ specifies that an instance of the event E should be detected in a state of the ADBS in which both *E1* and *E2* are present. On the other hand, $E = E1;E2$ specifies that an instance of the event E should be detected in a state in which the prior detection of *E1* is followed by a detection of *E2* (in that order). Clearly, one also needs an underlying detection mechanism for the expressions, for example, Petri Nets (17) or tree-like structures (5). Philosophically, the reason to incorporate both "E" and "C" parts of the ECA rules in ADBS is twofold: (1) It is intuitive to state that certain conditions should not always be checked but only upon the detection of certain events and (2) it is more cost-effective in actual implementations, as opposed to constant cycling through the set of rules.[2] Incorporating both events and conditions in the triggers has generated a plethora of different problems, such as the management of database state(s) during the execution of the triggers (18) and the binding of the detected events with the state(s) of the ADBS for the purpose of condition evaluation (19).

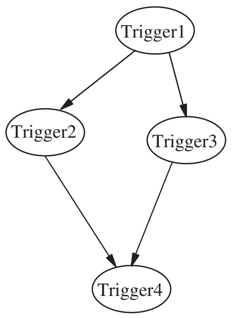
The need for formal characterization of the active rules (triggers) was recognized by the research community in the early 1990s. One motivation was caused by the observation that in different prototype systems [e.g., Postgres (4) vs. Starburst (2)], triggers with very similar syntactic structure would yield different executional behavior. Along with this was the need to perform some type of *reasoning* about the evolution of an active database system and to predict (certain aspects of) their behavior. As a simple example, given a set of triggers and a particular state of the DBMS, a database/application designer may wish to know whether a certain fact will hold in the database after a sequence of modifications (e.g., insertions, deletions, updates) have been performed. In the context of our example, one may be interested in the query *"will the average salary of the employees in the 'Shipping' department exceed 55K in any valid state which results via salary updates*." A translation of the active database specification into a logic program was proposed as a foundation for this type of reasoning in Ref. (20).

Two global properties that have been identified as desirable for any application of an ADBS are the *termination* and the *confluence* of a given set of triggers (21,22). The termination property ensures that for a given set of triggers in any initial state of the database and for any initial modification, the firing of the triggers cannot proceed indefinitely. On the other hand, the confluence property ensures that for a given set of triggers, in any initial state of the database and for any initial modification, the final state of the database is the same, regardless of the order of executing the (enabled) triggers. The main question is, given the specifications of a set of triggers, can one *statically*, (i.e., by applying some algorithmic techniques only to the triggers' specification) determine whether the properties of termination and/or confluence hold? To give a simple motivation, in many systems, the number of cascaded/recursive invocations of the triggers is bounded by a predefined constant to avoid infinite sequences of firing the triggers because of a particular event. Clearly, this behavior is undesirable, if the termination could have been achieved in a few more recursive executions of the triggers. Although run-time

[2] A noteworthy observation here is that the *occurrence* of a particular event is, strictly speaking, different from its *detection*, which is associated with a run-time processing cost.



(a) Non terminating cascading triggers



(b) Confluent triggers

Figure 1. Triggering graphs for termination and confluence.

termination analysis is a possible option, it is preferable to have static tools. In the earlier draft of the SQL3 standard, compile-time syntactic restrictions were placed on the triggers specifications to ensure termination/confluence. However, it was observed that these specifications may put excessive limitations on the expressive power on the triggers language, which is undesirable for many applications, and they were removed from the subsequent SQL99 draft.

For the most part, the techniques to analyze the termination and the confluence properties are based on labeled graph-based techniques, such as the triggering hyper graphs (22). For a simplified example, Fig. 1a illustrates a triggering graph in which the nodes denote the particular triggers, and the edge between two nodes indicates that the modifications generated by the action part of a given trigger node may generate the event that enables the trigger represented by the other node. If the graph contains a cycle, then it is possible for the set of triggers along that cycle to enable each other indefinitely through a cascaded sequence of invocations. In the example, the cycle is formed among Trigger1, Trigger3, Trigger4, and Trigger5. Hence, should Trigger1 ever become enabled because of the occurrence of its event, these four triggers could loop perpetually in a sequence of cascading firings. On the other hand, figure 1b illustrates a simple example of a confluent behavior of a set of triggers. When Trigger1 executes its action, both Trigger2 and Trigger3 are enabled. However, regardless of which one is selected for an execution, Trigger4 will be the next one that is enabled. Algorithms for static analysis of the ECA rules are presented in Ref. (21), which addresses their application to the triggers that conform to the SQL99 standard.

SEMANTIC DIMENSIONS OF ACTIVE DATABASES

Many of the distinctions among the various systems stem from the differences in the values chosen for a particular parameter (23). In some cases, the choice of that value is an integral part of the implementation ("hard wired"), whereas in other cases the ADBS provide a declarative syntax for the users to select a desired value. To better understand this concept, recall again our average salary maintenance scenario from the introduction. One of the possible sources that can cause the database to arrive at an undesired state is an update that increases the salary of a set of employees. We already illustrated the case of an insertion, now assume that the trigger that would correspond to the second case is specified as follows:

```
CREATE TRIGGER Update-Salary-Check
ON UPDATE OF Employee.Salary
IF (SELECT AVG Employee.Salary) > 65,000
UPDATE Employee
SET Employee.Salary = 0.95*Employee.Salary
```

Assume that it was decided to increase the salary of every employee in the "Maintenance" department by 10%, which would correspond to the following SQL statement:

```
UPDATE Employee
SET Employee.Salary = 1.10*Employee.Salary
WHERE Employee.Department = 'Maintenance'
```

For the sake of illustration, assume that three employees are in the "Maintenance" department, Bob, Sam, and Tom, whose salaries need to be updated. Strictly speaking, an update is essentially a sequence of a deletion of an old tuple followed by an insertion of that tuple with the updated values for the respective attribute(s); however, for the purpose of this illustration, we can assume that the updates execute atomically. Now, some obvious behavioral options for this simple scenario are:

- An individual instance of the trigger Update-Salary-Check may be fired immediately, for every single update of a particular employee, as shown in Fig. 2a.
- The DBMS may wait until all the updates are completed, and then execute the Update-Salary-Check, as illustrated in Fig. 2b.

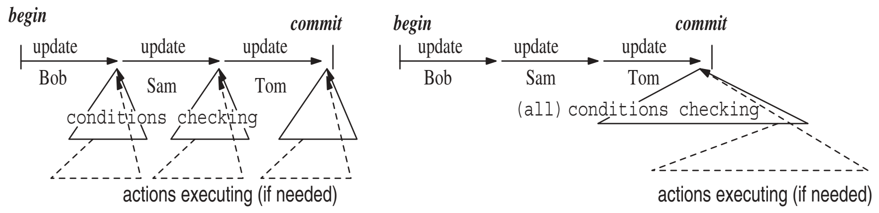


Figure 2. Different options for triggers execution.

- The DBMS waits for the completion of all the updates and the evaluation of the condition. If satisfied, the execution of the action for the instances of the Update-Salary-Check trigger may be performed either within the same transaction as the UPDATE Employee statement or in a different/subsequent transaction.

These issues illustrate some aspects that have motivated the researchers to identify various *semantic dimensions*, a term used to collectively identify the various parameters whose values may influence the executional behavior of ADBS. Strictly speaking, the model of the triggers' processing is coupled closely with the properties of the underlying DBMS, such as the data model, the transaction manager, and the query optimizer. However, some identifiable stages exist for every underlying DBMS:

- *Detection of events:* Events can be either *internal*, caused by a DBMS-invoked modification, transaction command, or, more generally, by any server-based action (e.g., clicking a mouse button on the display); or *external*, which report an occurrence of something outside the DBMS server (e.g., a humidity reading of a particular sensor, a location of a particular user detected by an RFID-based sensor, etc). Recall that (c.f. formalizing and reasoning), the events can be *primitive* or *composite*, defined in terms of the primitive events.
- *Detection of affected triggers:* This stage identifies the subset of the specified triggers, whose enabling events are among the detected events. Typically, this stage is also called the *instantiation* of the triggers.
- *Conditions evaluation:* Given the set of instantiated triggers, the DBMS evaluates their respective conditions and decides which ones are eligible for execution. Observe that the evaluation of the conditions may sometimes require a comparison between the values in the OLD (e.g., pretransaction state, or the state just before the occurence of the instantiating event) with the NEW (or current) state of the database.
- *Scheduling and execution:* Given the set of instantiated triggers, whose condition part is satisfied, this stage carries out their respective action-parts. In some systems [e.g., Starburst (2)] the users are allowed to specify a priority-based ordering among the triggers explicitly for this purpose. However, in the SQL standard (11), and in many commercially available DBMSs, the ordering is based on the time stamp of the creation of the trigger, and even this may not be enforced strictly at run-time. Recall (c.f. formalizing and reasoning) that the execution of the actions of the triggers may generate events that enable some other triggers, which causes a cascaded firing of triggers.

In the rest of this section, we present a detailed discussion of some of the semantic dimensions of the triggers.

Granularity of the Modifications

In relational database systems, a particular modification (insertion, deletion, update) may be applied to a single tuple or to a set of tuples. Similarly, in an object-oriented database system, the modifications may be applied to a single object or to a collection of objects—instances of a given class. Based on this distinction, the active rules can be made to react in a *tuple/instance* manner or in a *set-oriented* manner. An important observation is that this type of granularity is applicable in two different places in the active rules: (1) the events to which a particular rule reacts (is "awoken" by) and (2) the modifications executed by the action part of the rules. Typically, in the DBMS that complies with the SQL standard, this distinction is specified by using FOR EACH ROW (tuple-oriented) or FOR EACH STATEMENT (set-oriented) specification in the respective triggers. In our motivational scenario, if one would like the trigger Update-Salary to react to the modifications of the individual tuples, which correspond to the behavior illustrated in Fig. 2a, its specification should be:

```
CREATE TRIGGER Update-Salary-Check
ON UPDATE OF Employee.Salary
FOR EACH ROW
IF SELECT(...)
...
```

Coupling Among Trigger's Components

Because each trigger that conforms to the ECA paradigm has three distinct parts—the Event, Condition, and Action—one of the important questions is how they are synchronized. This synchronization is often called the *coupling* among the triggers components.

- *E-C coupling:* This dimension describes the temporal relationship among the events that enable certain triggers and the time of evaluating their conditions parts, with respect to the transaction in which the events were generated. With *immediate coupling*, the conditions are evaluated as soon as the basic modification that produced the events is completed. Under the *delayed coupling* mode, the evaluation of the conditions is delayed until a specific point (e.g., a special "event" takes place such as an explicit rule-processing point in the transaction). Specifically, if this special event is the attempt to *commit* the transaction, then the coupling is also called *deferred*.
- *C-A coupling:* Similarly, for a particular trigger, a temporal relationship exists between the evaluation of its condition and (if satisfied) the instant its action is executed. The options are the same as for the *E-C* coupling: *immediate*, in which case the action is executed as soon as the condition-evaluation is completed (in case it evaluates to true); *delayed,* which executes the action at some special point/event; and *deferred*, which is the case when the actions are executed at the end (just before *commit*) of the transaction in which the condition is evaluated.

A noteworthy observation at this point is that the semantic dimensions should not be understood as isolated completely from each other but, to the contrary, their values may be correlated. Among the other reasons, this correlation exists because the triggers manager cannot be implemented in isolation from the query optimizer and the transaction manager. In particular, the coupling modes discussed above are not independent from the transaction processing model and its relationship with the individual parts of the triggers. As another semantic dimension in this context, one may consider whether the conditions evaluation and the actions executions should be executed in the same transaction in which the triggering events have occurred (note that the particular transaction may be aborted because of the effects of the triggers processing). In a typical DBMS setting, in which the ACID (atomicity, consistency, isolation, and durability) properties of the transactions must be ensured, one would like to maintain the conditions evaluations and actions executions within the same transaction in which the triggering event originated. However, if a more sophisticated transaction management is available [e.g., nested transactions (24)] they may be processed in a separate subtransaction(s), in which the failure of a subtransaction may cause the failure of a parent transaction in which the events originated, or in two different transactions. This transaction is known commonly as a *detached* coupling mode.

Events Consumption and Composition

These dimensions describe how a particular event is treated when processing a particular trigger that is enabled due to its occurrence, as well as how the impact of the net effect of a set of events is considered.

One of the differences in the behavior of a particular ADBS is caused by the selection of the *scope*(23) of the event consumptions:

- *NO consumption:* the evaluation and the execution of the conditions part of the enabled/instantiated triggers have no impact on the triggering event. In essence, this means that the same event can enable a particular trigger over and over again. Typically, such behavior is found in the production rule systems used in expert systems (15).
- *Local consumption:* once an instantiated trigger has proceeded with its condition part evaluation, that trigger can no longer be enabled by the same event. However, that particular event remains eligible for evaluation of the condition the other triggers that it has enabled. This feature is the most common in the existing active database systems. In the setting of our motivational scenario, assume that we have another trigger, for example, Maintain-Statistics, which also reacts to an insertion of new employees by increasing properly the total number of the hired employees in the respective departmental relations. Upon insertion of a set of new employees, both New-Employee-Salary-Check and Maintain-Statistics triggers will be enabled. Under the local consumption mode, in case the New-Employee-Salary-Check trigger executes first, it is no longer enabled by the same insertion. The Maintain-Statistics trigger, however, is left enabled and will check its condition and/or execute its action.
- *Global consumption:* Essentially, global consumption means that once the first trigger has been selected for its processing, a particular event can no longer be used to enable any other triggers. In the settings of our motivational scenario, once the given trigger New-Employee-Salary-Check has been selected for evaluation of its condition, it would also disable the Maintain-Statistics despite that it never had its condition checked. In general, this type of consumption is appropriate for the settings in which one can distinguish among "regular" rules and "exception" rules that are enabled by the same event. The "exception" not only has a higher priority, but it also disables the processing of the "regular" rule.

A particular kind of event composition, which is encountered in practice, frequently is the *event net effect*. The basic distinction is whether the system should consider the impact of the occurrence of a particular event, regardless of what are the subsequent events in the transaction, or consider the possibility of invalidating some of the events that have occurred earlier. As a particular example, the following intuitive policy for computing the net effects has been formalized and implemented in the Starburst system (2):

- If a particular tuple is created (and possibly updated) in a transaction, and subsequently deleted within that same transaction, the net effect is *null*.

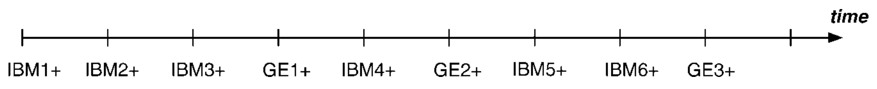


Figure 3. Composite events and consumption.

- If a particular tuple is created (respectively, updated) in a transaction, and that tuple is updated subsequently several times, the net effect is the creation of the final version of that tuple (respectively, the single update equivalent to the final value).
- If a particular tuple is updated and deleted subsequently in a given transaction, then the net effect is the deletion of the original tuple.

Combining the computation of the net effects in systems that allow specification of composite events via an event algebra (5,17) is a very complex problem. The main reason is that in a given algebra, the detection of a particular composite event may be in a state in which several different instances of one of its constituent events have occurred. Now, the question becomes what is the policy for *consuming* the primitive events upon a detection of a composite one. An illustrative example is provided in Fig. 3. Assume that the elementary (primitive) events correspond to tickers from the stockmarket and the user is interested in the composite event: *CE = (two consecutive increases of the IBM stock) AND (two consecutive increases of the General Electric [GE] stock).* Given the timeline for the sequence of events illustrated in Fig. 3, upon the second occurrence of the GE stock increase (GE2+), the desired composite event CE can be detected. However, now the question becomes *which* of the primitive events should be used for the detection of CE (6 ways exist to couple IBM-based events), and *how* should the rest of the events from the history be consumed for the future (e.g., if GE2+ is not consumed upon the detection of CE, then when GE3+ occurs, the system will be able to detect another instance of CE). Chakravarthy et al. (5) have identified four different contexts (*recent, chronicle, continuous,* and *cumulative*) of consuming the earlier occurrences of the primitive constituent events which enabled the detection of a given composite event.

Data Evolution

In many ADBSs, it is important to query the history concerning the execution of the transaction(s). For example, in our motivational scenario, one may envision a modified constraint that states that the average salary increase in the enterprise should not exceed 5% from its previous value when new employees are and/or inserted when the salaries of the existing employees are updated. Clearly, in such settings, the conditions part of the respective triggers should compare the current state of the database with the older state.

When it comes to past database states, a special syntax is required to specify properly the queries that will retrieve the correct information that pertains to the prior database states. It can be speculated that every single state that starts from the *begin* point of a particular transaction should be available for inspection; however, in practice, only a few such states are available (c.f. Ref. (23)):

- *Pretransaction state:* the state of the database just before the execution of the transaction that generated the enabling event.
- *Last consideration state:* given a particular trigger, the state of the database after the last time that trigger has been considered (i.e., for its condition evaluation).
- *Pre-event state:* given a particular trigger, the state of the database just before the occurrence of the event that enabled that trigger.

Typically, in the existing commercially available DBMS that offers active capabilities, the ability to query the past states refers to the pretransaction state. The users are given the keywords OLD and NEW to specify declaratively which part needs to be queried when specifying the condition part of the triggers (11).

Another option for inspecting the history of the active database system is to query explicitly the set of occurred events. The main benefit of this option is the increased flexibility to specify the desired behavioral aspects of a given application. For example, one may wish to query not all the items affected by a particular transaction, but only the ones that participated in the generation of the given composite event that enabled a particular trigger (5). Some prototype systems, [e.g., Chimera (25) offer this extended functionality, however, the triggers in the commercially available DBMS that conform to the SQL standard are restricted to querying the database states only (c.f., the OLD and NEW above).

Recent works (26) have addressed the issues of extending the capabilities of the commercially available ORDBMS Oracle 10g (27) with features that add a flexibility for accessing various portions (states) of interest throughout the evolution of the ADBS, which enable sophisticated management of events for wide variety of application domains.

Effects Ordering

We assumed that the execution of the action part of a particular trigger occurs not only *after* the occurrence of the event, but also after the effects of executing the modifications that generated that event have been incorporated. In other words, the effects of executing a particular trigger were adding to the effects of the modifications that were performed by its enabling event. Although this seems to be the most intuitive approach, in some applications, such as alerting or security monitoring, it may be desirable to have the action part of the corresponding trigger execute *before* the modifications of the events take place, or even *instead* of the modifications.

Typical example is a trigger that detects when an unauthorized user has attempted to update the value of a particular tuple in a given relation. Before executing the user's request, the respective log-file needs to be updated properly. Subsequently, the user-initiated transaction must be aborted; instead, an alert must be issued to the database administrator. Commercially available DBMS offer the flexibility of stating the BEFORE, AFTER, and INSTEAD preferences in the specification of the triggers.

Conflict Resolution and Atomicity of Actions

We already mentioned that if more than one trigger is enabled by the occurrence of a particular event, some selection must be performed to evaluate the respective conditions and/or execute the actions part. From the most global perspective, one may distinguish between the *serial* execution, which selects a single rule according to a predefined policy, and a *parallel* execution of all the enabled triggers. The latter was envisioned in the HiPAC active database systems (c.f. Ref. (28)) and requires sophisticated techniques for concurrency management. The former one can vary from specifying the total priority ordering completely by the designer, as done in the Postgres system (4), to partial ordering, which specifies an incomplete precedence relationship among the triggers, as is the case in the Starburst system (20). Although the total ordering among the triggers may enable a deterministic behavior of the active database, it may be too demanding on the designer, who always is expected to know exactly the intended behavior of all the available rules (23). Commercial systems that conform with the SQL99 standard do not offer the flexibility of specifying an ordering among the triggers. Instead, the default ordering is by the timestamp of their creation.

When executing the action part of a given trigger, a particular modification may constitute an enabling event for some other trigger, or even for a new instance of the same trigger whose action's execution generated that event. One option is to interrupt the action of the currently executing trigger and process the triggers that were "awoken" by it, which could result in cascaded invocation where the execution of the trigger that produced the event is suspended temporarily. Another option is to ignore the occurrence of the generated event temporarily, until the action part of the currently executing trigger is completed (atomic execution). This action illustrates that the values in different semantic dimensions are indeed correlated. Namely, the choice of the atomicity of the execution will impact the value of the E-C/C-A coupling modes: one cannot expect an immediate coupling if the execution of the actions is to be atomic.

Expressiveness Issues

As we illustrated, the choice of values for a particular semantic dimension, especially when it comes to the relationship with the transaction model, may yield different outcomes of the execution of a particular transaction by the DBMS (e.g., deferred coupling will yield different behavior from the immediate coupling). However, another subtle aspect of the active database systems is dependent strongly on their chosen semantic dimensions – the expressive power. Picouet and Vianu (29) introduced a broad model for active databases based on the unified framework of relational Turing machines. By restricting some of the values of the subset of the semantic dimensions and thus capturing the interactions between the sequence of the modifications and the triggers, one can establish a yardstick to compare the expressive powers of the various ADBSs. For example, it can be demonstrated that:

- The A-RDL system (30) under the immediate coupling mode is equivalent to the Postgres system (4) on ordered databases.
- The Starburst system (2) is incomparable to the Postgres system (4).
- The HiPAC system (28) subsumes strictly the Starburst (2) and the Postgres (4) systems.

Although this type of analysis is extremely theoretical in nature, it is important because it provides some insights that may have an impact on the overall application design. Namely, when the requirements of a given application of interest are formalized, the knowledge of the expressive power of the set of available systems may be a crucial factor to decide which particular platform should be used in the implementation of that particular application.

OVERVIEW OF ACTIVE DATABASE SYSTEMS

In this section, we outline briefly some of the distinct features of the existing ADBS—both prototypes as well as commercially available systems. A detailed discussion of the properties of some systems will also provide an insight of the historic development of the research in the field of ADBS, can be found in Refs. (1) and (7).

Relational Systems

A number of systems have been proposed to extend the functionality of relational DBMS with active rules. Typically, the events in such systems are mostly database modifications (insert, delete, update) and the language to specify the triggers is based on the SQL.

- *Ariel* (31): The Ariel system resembles closely the traditional *Condition* $\rightarrow$ *Action* rules from expert systems literature (15), because the specification of the Event part is optional. Therefore, in general, NO event consumption exists, and the coupling modes are immediate.
- *Starburst* (2): This system has been used extensively for database-internal applications, such as integrity constraints and views maintenance. Its most notable features include the set-based execution model and the introduction of the net effects when considering the modifications that have led to the occurrence of a particular event. Another particular feature

introduced by the Starburst system is the concept of *rule processing points*, which may be specified to occur during the execution of a particular transaction or at its end. The execution of the action part is atomic.

- *Postgres* (4): The key distinction of Postgres is that the granularity of the modifications to which the triggers react is tuple (row) oriented. The coupling modes between the E-C and the C-A parts of the triggers are immediate and the execution of the actions part is interruptable, which means that the recursive enabling of the triggers is an option. Another notable feature of the Postgres system is that it allows for INSTEAD OF specification in its active rules.

Object-Oriented Systems

One of the distinct features of object-oriented DBMS (OODBMS) is that it has *methods* that are coupled with the definition of the classes that specify the structure of the data objects stored in the database. This feature justifies the preference for using OODBMS for advanced application domains that include extended behavior management. Thus, the implementation of active behavior in these systems is coupled tightly with a richer source of events for the triggers (e.g., the execution of any method).

- *ODE* (32): The ODE system was envisioned as an extension of the C++ language with database capabilities. The active rules are of the C-A type and are divided into constraints and triggers for the efficiency of the implementations. Constraints and triggers are both defined at a class level and are considered a property of a given class. Consequently, they can be inherited. One restriction is that the updates of the individual objects, caused by private member functions, cannot be monitored by constraints and triggers. The system allows for both immediate coupling (called *hard constraints*) and deferred coupling (called *soft constraints*), and the triggers can be declared as executing once-only or perpetually (reactivated).
- *HiPAC* (28): The HiPAC project has pioneered many of the ideas that were used subsequently in various research results on active database systems. Some of the most important contributions were the introduction of the coupling modes and the concept of composite events. Another important feature of the HiPAC system was the extension that provided the so called *delta-relation*, which monitors the net effect of a set of modifications and made it available as a part of the querying language. HiPAC also introduced the visionary features of parallel execution of multiple triggers as subtransactions of the original transaction that generated their enabling events.
- *Sentinel* (5): The Sentinel project provided an active extension of the OODBMS, which represented the active rules as database objects and focused on the efficient integration of the rule processing module within the transaction manager. One of the main novelties discovered this particular research project was the introduction of a rich mechanism for to specify and to detect composite events.
- *SAMOS* (18): The SAMOS active database prototype introduced the concept of an *interval* as part of the functionality needed to manage composite events. A particular novelty was the ability to include the monitoring intervals of interest as part of the specification of the triggers. The underlying mechanism to detect the composite events was based on Colored Petri-Nets.
- *Chimera* (25): The Chimera system was envisioned as a tool that would seamlessly integrate the aspects of object orientation, deductive rules, and active rules into a unified paradigm. Its model has strict underlying logical semantics (fixpoint based) and very intuitive syntax to specify the active rules. It is based on the EECA (Extended-ECA) paradigm, specified in Ref. (23), and it provides the flexibility to specify a wide spectrum of behavioral aspects (e.g., semantic dimensions). The language consists of two main components: (1) declarative, which is used to specify queries, deductive rules, and conditions of the active rules; and (2) procedural, which is used to specify the nonelementary operations to the database, as well as the action parts of the triggers.

Commercially Available Systems

One of the earliest commercially available active database systems was DB2 (3), which integrated trigger processing with the evaluation and maintenance of declarative constraints in a manner fully compatible with the SQL92 standard. At the time it served as a foundation model for the draft of the SQL3 standard. Subsequently, the standard has migrated to the SQL99 version (11), in which the specification of the triggers is as follows:

```
<trigger definition> ::=
  CREATE TRIGGER <trigger name>
  {BEFORE | AFTER} <trigger event> ON
    <table name>
  [REFERENCING <old or new values alias list>]
  [FOR EACH {ROW | STATEMENT}]
  [<trigger condition>]
  <trigger action>
<trigger event> ::= INSERT | DELETE | UPDATE
    [OF <column name list>]
<old or new values alias list> ::= {OLD | NEW}
   [AS] <identifier> | {OLD_TABLE |
        NEW_TABLE} [AS] <identifier>
```

The condition part in the SQL99 triggers is optional and, if omitted, it is considered to be true; otherwise, it can be any arbitrarily complex SQL query. The action part, on the other hand, is any sequence of SQL statements, which includes the invocation of stored procedures, embedded within a single BEGIN – END block. The only statements that are excluded from the available actions pertain to connections, sessions, and transactions processing.

Commercially available DBMS, with minor variations, follow the guidelines of the SQL99 standards.

In particular, the Oracle 10g (27), an object-relational DBMS (ORDBMS), not only adheres to the syntax specifications of the SQL standard for triggers (28), but also provides some additions: The triggering event can be specified as a logical disjunction (ON INSERT OR UPDATE) and the INSTEAD OF option is provided for the action's execution. Also, some system events (startup/shutdown, server error messages), as well as user events (logon/logoff and DDL/DML commands), can be used as enabling events in the triggers specification. Just like in the SQL standard, if more than one trigger is enabled by the same event, the Oracle server will attempt to assign a priority for their execution based on the timestamps of their creation. However, it is not guarantees that this case will actually occur at run time. When it comes to dependency management, Oracle 10g server treats triggers in a similar manner to the stored procedures: they are inserted automatically into the data dictionary and linked with the referenced objects (e.g., the ones which are referenced by the action part of the trigger). In the presence of integrity constraints, the typical executional behavior of the Oracle 10g server is as follows:

1. Run all BEFORE *statement* triggers that apply to the statement.
2. Loop for each row affected by the SQL statement.
   a. Run all BEFORE *row* triggers that apply to the statement.
   b. Lock and change row, and perform integrity constraint checking. (The lock is not released until the transaction is committed.)
   c. Run all AFTER *row* triggers that apply to the statement.
3. Complete deferred integrity constraint checking.
4. Run all AFTER *statement* triggers that apply to the statement.

The Microsoft Server MS-SQL also follows closely the syntax prescribed by the SQL99 standard. However, it has its own additions; for example, it provides the INSTEAD OF option for triggers execution, as well as a specification of a restricted form of composite events to enable the particular trigger. Typically, the statements execute in a tuple-oriented manner for each row. A particular trigger is associated with a single table and, upon its definition, the server generates a virtual table automatically for to access the old data items. For example, if a particular trigger is supposed to react to INSERT on the table Employee, then upon insertion to Employee, a virtual relation called Inserted is maintained for that trigger.

NOVEL CHALLENGES FOR ACTIVE RULES

We conclude this article with a brief description of some challenges for the ADBSs in novel application domains, and with a look at an extended paradigm for declaratively specifying reactive behavior.

Application Domains

Workflow management systems (WfMS) provide tools to manage (modeling, executing, and monitoring) *workflows*, which are viewed commonly as processes that coordinate various cooperative activities to achieve a desired goal. Workflow systems often combine the *data centric* view of the applications, which is typical for information systems, with their *process centric* behavioral view. It has already been indicated (12) that WfMS could benefit greatly by a full use of the tools and techniques available in the DBMS when managing large volumes of data. In particular, Shankar et al. (12) have applied active rules to the WfMS settings, which demonstrates that data-intensive scientific workflows can benefit from the concept of *active tables* associated with the programs. One typical feature of workflows is that many of the activities may need to be executed by distributed agents (*actors* of particular *roles*), which need to be synchronized to optimize the concurrent execution. A particular challenge, from the perspective of triggers management in such distributed settings, is to establish a common (e.g., transaction-like) context for their main components—events, conditions, and actions. As a consequence, the corresponding triggers must execute in a detached mode, which poses problems related not only to the consistency, but also to their efficient scheduling and execution (33).

Unlike traditional database applications, many novel domains that require the management of large quantities of information are characterized by the high volumes of data that arrive very fast in a stream-like fashion (8). One of the main features of such systems is that the queries are no longer instantaneous; they become *continuous*/*persistent* in the sense that users expect the answers to be updated properly to reflect the current state of the streamed-in values. Clearly, one of the main aspects of the continuous queries (CQ) management systems is the ability to react quickly to the changes caused by the variation of the streams and process efficiently the modification of the answers. As such, the implementation of CQ systems may benefit from the usage of the triggers as was demonstrated in the Niagara project (9). One issue related to the scalability of the CQ systems is the very scalability of the triggers management (i.e., many instances of various triggers may be enabled). Although it is arguable that the problem of the scalable execution of a large number of triggers may be coupled closely with the nature of the particular application domain, it has been observed that some general aspects of the scalability are applicable universally. Namely, one can identify similar predicates (e.g., in the conditions) across many triggers and group them into equivalence classes that can be indexed on those predicates. This project may require a more involved system catalog (34), but the payoff is a much more efficient execution of a set of triggers. Recent research has also demonstrated that, to capture the intended semantics of the application domain in dynamic environments, the events may have to be assigned an interval-based semantics (i.e., duration may need to be associated with their detection). In particular, in Ref. (35), the authors have demonstrated that if the commonly accepted instantaneous semantics for events occurrence is used in traffic management settings, one may obtain an unintended meaning for the composite events.

Moving objects databases (MODs) are concerned with the management of large volumes of data that pertain to the location-in-time information of the moving entities, as well as efficient processing of the spatio-temporal queries that pertain to that information (13). By nature, MOD queries are continuous and the answers to the pending queries change because of the changes in the location of the mobile objects, which is another natural setting for exploiting an efficient form of a reactive behavior. In particular, Ref. (14) proposed a framework based on the existing triggers in commercially available systems to maintain the correctness of the continuous queries for trajectories. The problem of the scalable execution of the triggers in these settings occurs when a traffic abnormality in a geographically small region may cause changes to the trajectories that pass through that region and, in turn, invalidate the answers to spatio-temporal queries that pertain to a much larger geographic area. The nature of the continuous queries' maintenance is dependent largely on the model adopted for the mobility representation, and the MOD-field is still very active in devising efficient approaches for the queries management which, in one way or another, do require some form of active rules management.

Recently, the wireless sensor networks (WSNs) have opened a wide range of possibilities for novel applications domains in which the whole process of gathering and managing the information of interest requires new ways of perceiving the data management problems (36). WSN consist of hundreds, possibly thousands, of low-cost devices (sensors) that are capable of measuring the values of a particular physical phenomenon (e.g., temperature, humidity) and of performing some elementary calculations. In addition, the WSNs are also capable of communicating and self-organizing into a network in which the information can be gathered, processed, and disseminated to a desired location. As an illustrative example of the benefits of the ECA-like rules in WSN settings, consider the following scenario (c.f. Ref. (6)): whenever the sensors deployed in a given geographic area of interest have detected that the average level of carbon monoxide in the air over any region larger than 1200 ft^2 exceeds 22%, an alarm should be activated. Observe that here the event corresponds to the updates of the (readings of the) individual sensors; the condition is a continuous query evaluated over the entire geographic zone of interest, and with a nested sub-query of identifying the potentially-dangerous regions. At intuitive level, this seems like a straightforward application of the ECA paradigm. Numerous factors in sensor networks affect the efficient implementation of this type of behavior: the energy resource of the individual nodes is very limited, the communication between nodes drains more current from the battery than the sensing and local calculations, and unlike the traditional systems where there are few vantage points to generate new events, in WSN settings, any sensor node can be an event-generator. The detection of composite events, as well as the evaluation of the conditions, must to be integrated in a fully distributed environment under severe constraints (e.g., energy-efficient routing is a paramount). Efficient implementation of the reactive behavior in a WSN-based databases is an ongoing research effort.

The $(ECA)^2$ Paradigm

Given the constantly evolving nature of the streaming or moving objects data, along with the consideration that it may be managed by distributed and heterogeneous sources, it is important to offer a declarative tool in which the users can actually specify how the triggers themselves should evolve. Users can adjust the events that they monitor, the conditions that they need to evaluate, and the action that they execute. Consider, for example, a scenario in which a set of motion sensors deployed around a region of interest is supposed to monitor whether an object is moving continuously toward that region for a given time interval. Aside from the issues of efficient detection of such an event, the application may require an alert to be issued when the status of the closest air field is such that fewer than a certain number of fighter jets are available. In this setting, both the event detection and the condition evaluation are done in distributed manner and are continuous in nature. Aside from the need of their efficient synchronization, the application demands that when a particular object ceases to move continuously toward the region, the condition should not be monitored any further for that object. However, if the object in question is closer than a certain distance (after moving continuously toward the region of interest for a given time), in turn, another trigger may be enabled, which will notify the infantry personnel. An approach for declarative specification of triggers for such behavior was presented in Ref. (37) where the $(ECA)^2$ paradigm (evolving and context-aware event-condition-action) was introduced. Under this paradigm, for a given trigger, the users can embed children triggers in the specifications, which will become enabled upon the occurrences of certain events in the environment, and only when their respective parent triggers are no longer of interest. The children triggers may consume their parents either completely, by eliminating them from any consideration in the future or partially, by eliminating only the particular instance from the future consideration, but allowing a creation of subsequent instances of the parent trigger. Obviously, in these settings, the coupling modes among the E-C and C-A components of the triggers must to be detached, and for the purpose of their synchronization the concept of meta-triggers was proposed in Ref. (37). The efficient processing of such triggers is still an open challenge.

BIBLIOGRAPHY

1. J. Widom and S. Ceri, *Active Database Systems: Triggers and Rules for Advanced Database Processing*, San Francisco: Morgan Kauffman, 1996.
2. J. Widom, The Starburst Active Database Rule System, *IEEE Trans. Knowl. Data Enginee.*, **8**(4): 583–595, 1996.
3. R. Cochrane, H. Pirahesh and N. M. Mattos, *Integrating Triggers and Declarative Constraints in SQL Database Systems*, International Conference on Very Large Databases, 1996.

4. M. Stonebraker, The integration of rule systems and database systems, *IEEE Trans. Knowl. Data Enginee.*, **4**(5): 416–432, 1992.
5. S. Chakravarthy, V. Krishnaprasad, E. Answar, and S. K. Kim, *Composite Events for Active Databases: Semantics, Contexts and Detection*, International Conference on Very Large Databases (VLDB), 1994.
6. M. Zoumboulakis, G. Roussos, and A. Poulovassilis, *Active Rules for Wireless Networks of Sensors & Actuators*, International Conference on Embedded Networked Sensor Systems, 2003.
7. N. W. Paton, *Active Rules in Database Systems*, New York: Springer Verlag, 1999.
8. B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, *Models and Issues in Data Stream Systems*, International Conference on Principles of Database Systems, 2002.
9. J. Chen, D.J. DeWitt, F. Tian, and Y. Wang, *NiagaraCQ: A Scalable Continuous Query System for Internet Databases*, ACM SIGMOD International Conference on Management of Data, 2000.
10. A. Carzaniga, D. Rosenblum, and A. Wolf, *Achieving Scalability and Expressiveness in an Internet-scale Event Notification Service,* ACM Symposium on Principles of Distributed Computing, 2000.
11. ANSI/ISO International Standard: Database Language SQL. Available: http://webstore.ansi.org.
12. S. Shankar, A. Kini, D. J. DeWitt, and J. F. Naughton, Integrating databases and workflow systems, *SIGMOD Record*, **34**(3): 5–11, 2005.
13. R.H. Guting and M. Schneider, *Moving Objects Databases*, San Francisco: Morgan Kaufman, 2005.
14. G. Trajcevski and P. Scheuermann, Reactive maintenance of continuous queries, *Mobile Comput. Commun. Rev.*, **8**(3): 22–33, 2004.
15. L. Brownston, K. Farrel, E. Kant, and N. Martin, *Programming Expert Systems in OPS5: An Introduction to Rule-Base Programming*, Boston: Addison-Wesley, 2005.
16. A.P. Sistla and O. Wolfson, *Temporal Conditions and Integrity Constraints in Active Database Systems ACM SIGMOD*, International Conference on Management of Data, 1995.
17. S. Gatziu and K. R. Ditrich, *Events in an Active Object-Oriented Database System,* International Workshop on Rules in Database Systems, 1993.
18. K. Dittrich, H. Fritschi, S. Gatziu, A. Geppert, and A. Vaduva, SAMOS in hindsight: Experiences in building an active object-oriented DBMS, *Informat. Sys.*, **30**(5): 369–392, 2003.
19. S.D. Urban, T. Ben-Abdellatif, S. Dietrich, and A. Sundermier, Delta abstractions: a technique for managing database states in runtime debugging of active database rules, *IEEE-TKDE*, **15**(3): 597–612, 2003.
20. C. Baral, J. Lobo, and G. Trajcevski, *Formal Characterization of Active Databases: Part II,* International Conference on Deductive and Object-Oriented Databases (DOOD), 1997.
21. E. Baralis and J. Widom, An algebraic approach to static analysis of active database rules, *ACM Trans. Database Sys.*, **27**(3): 289–332, 2000.
22. S.D. Urban, M.K. Tschudi, S.W. Dietrich, and A.P. Karadimce, Active rule termination analysis: an implementation and evaluation of the refined triggering graph method, *J. Intell. Informat. Sys.*, **14**(1): 29–60, 1999.
23. P. Fraternali and L. Tanca, A structured approach for the definition of the semantics of active databases, *ACM Trans. Database Sys.*, **22**(4): 416–471, 1995.
24. G. Weikum and G. Vossen, *Transactional Information Systems: Theory, Algorithms and the Practice of Concurrency Control*, San Francisco: Morgan Kauffman, 2001.
25. P. Fraternali and S. Parabochi, Chimera: a language for designing rule applications, in: N. W. Paton (ed.), *Active Rules in Database System*, Berlin: Springer-Verlog, 1999.
26. M. Thome, D. Gawlick, and M. Pratt, *Event Processing with an Oracle Database*, SIGMOD International Conference on Management of Data, 2005.
27. K. Owens, *Programming Oracle Triggers and Stored Procedures*, (3rd ed.), O'Reily Publishers, 2003.
28. U. Dayal, A.P. Buchmann, and S. Chakravarthy, The HiPAC project, in J. Widom and S. Ceri, *Active Database Systems. Triggers and Rules for Advanced Database Processing*, San Francisco: Morgan Kauffman, 1996.
29. P. Picouet and V. Vianu, Semantics and expressiveness issues in active databases, *J. Comp. Sys. Sci.*, **57**(3): 327–355, 1998.
30. E. Simon and J. Kiernan, The A-RDL system, in J. Widom and S. Ceri, *Active Database Systems: Triggers and Rules for Advanced Database Processing*, San Francisco: Morgan Kauffman, 1996.
31. E. N. Hanson, *Rule Condition Testing and Action Execution in Ariel*, ACM SIGMOD International Conference on Management of Data, 1992.
32. N. Gehani and H.V. Jagadish, *ODE as an Active Database: Constraints and Triggers*, International Conference on Very Large Databases, 1992.
33. S. Ceri, C. Gennaro, S. Paraboschi, and G. Serazzi, Effective scheduling of detached rules in active databases, *IEEE Trans. Knowl. Data Enginee.*, **16**(1): 2–15, 2003.
34. E.N. Hanson, S. Bodagala, and U. Chadaga, Trigger condition testing and view maintenance using optimized discrimination networks, *IEEE Trans. Know. Data Enginee.*, **16**(2): 281–300, 2002.
35. R. Adaikkalvan and S. Chakravarthy, *Formalization and Detection of Events Using Interval-Based Semantics*, International Conference on Advances in Data Management, 2005.
36. F. Zhao and L. Guibas, *Wireless Sensor Networks: An Information Processing Approach*, San Francisco: Morgan Kaufmann, 2004.
37. G. Trajcevski, P. Scheuermann, O. Ghica, A. Hinze, and A. Voisard, *Evolving Triggers for Dynamic Environments*, International Conference on Extending the Database Technology, 2006.

PETER SCHEUERMANN
GOCE TRAJCEVSKI
Northwestern University
Evanston, Illinois