Issues in Active Databases

James A. Bailey
Department of Computer Science
University of Melbourne
Melbourne
Australia

jbailey@cs.mu.oz.au

Kotagiri Ramamohanarao
Department of Computer Science
University of Melbourne
Melbourne
Australia

rao@cs.mu.oz.au

Abstract

This paper examines an emerging technology known as active databases. The focus is on issues which need to be considered in their development and the way they have been addressed in currently implemented systems. We examine what defines active behavior, why it is useful, the semantics of its specification and choices that have to be made when realising it in an actual system. We see that a multitude of options exist when designing an active database and future research will prove to be both necessary and fruitful.

1 Introduction

Traditional database systems provide a mechanism for storing large amounts of data and an interface for manipulating and querying this data. They are, however, passive in the sense that their state can only change as a result of outside influences. In contrast, an active database is a system providing the functionality of a traditional database and additionally is capable of reacting automatically to state changes, both internal and external, without user intervention. The rules which define this active behaviour are known as triggers or production rules. The terms triggers and rules will be used interchangeably for the remainder of this paper.

Applications of active databases are many and varied. They can range from handling simple integrity constraints to performing complex database changes. Some possible applications are

- gathering statistics on database usage, measuring performance and providing access control
- automatically detecting the insertion of an invalid data item in the database and initiating operations to restore consistency (Ceri [6])
- maintaining materialised views (both incrementally and non incrementally) See Ceri [7] and Harrison [19].
- heterogeneous database integration. Maintaining consistency across a number of overlapping databases (Ceri [8])
- rule based inferencing (as in expert systems)
- setting defaults
- user notification of appropriate conditions
- providing inheritance mechanisms for object oriented data models
- allowing indirect communication of applications (i.e via triggers)

It can be argued that these kind of functions could be performed equally well by a passive database coupled to an application program. To achieve this in such a system, one would normally have the application program poll the database at defined intervals to determine whether a particular condition is true (Chakravarthy [9]). This suffers from the drawback that if the polling interval is too low, then a critical change may not be detected quickly enough and if it is too high, then efficiency is lost. There will also always be some overhead in moving data between the two systems. Polling has implications from a software engineering point of view, since it puts the burden of developing the polling mechanism on the application developer. Software modularity can be compromised, since any modification to the situations being monitored will require modification of all the applications which poll the database. It may also be hard to do optimisation for situation monitoring. An active system has the advantage that one can be sure that the rules and the data are consistent, because one system is administering both types of entities. With polling, there will always be some potential for data mismatch.

Active databases might seem superficially similar to expert systems (Baralis [3] and Segev [22]). They are, however, different in certain respects. Expert systems are generally special purpose, single user systems and so there is little concern about sharing resources with other users. Their execution model is fundamentally different, expert system rules are executed upon an explicit request for information, while triggers are executed as side effects. Finally, since expert system rules are used for inferencing, their execution order is often not important. In an active database, this order can produce varying results. The remainder of this paper will be devoted to examining the types of issues that need to be considered when designing an active database system. They are

- The architecture of the system used to incorporate triggers (section 2)
- The language used to specify triggers (section 3)
- How triggers should be executed (section 4)
- The order in which triggers can be activated or selected (section 5)
- The effect of nested trigger activation (section 7)
- Methods for optimising trigger execution (section 6)
- Management of a collection of triggers (section 8)

These issues will be examined within the context of existing systems. Not much emphasis will be put on the type of data modelling used (relational vs object oriented). The other main research area which deals with rules in database systems is deductive databases. Deductive databases use logic programming style rules to provide a powerful query language. In Widom [29] it is argued that deductive style languages give a higher level of abstraction than active style ones and that active languages tend to provide more 'features' than deductive ones. The features currently offered by active databases will become more apparent when we look at some specific systems.

2 Active System Architectures

When building an active database, a decision has to be made about the type of architecture used. To do this, we need to extract out what are the key features of an active system. The primary one is the ability to manage the triggering rules. As we will see in section 3, although rules have separate components, they are closely related and it makes sense to have a designated authority administering all of them together. This can be kept separate from the rest of the DBMS or integrated within. It will be in charge of executing and optimising rules(see section 6). In order for rules to be executed, allowance must be made within the transaction structure for their presence and there must be some scheme for ordering them. We will examine this more closely in sections 4 and 5. Communication with this rule manager by users should be possible, so that triggers can be defined and deleted (section 8).

So we have three broad categories of architecture - Chakravarthy [9]

- Application based architecture: This has busy waiting or polling built in as part of the application.
- Layered architecture: One builds a layer on top of a passive DBMS. All operations using active functionality must communicate through this layer (Schreier [23]).
- Integrated architecture: The core functionality of a passive DBMS is expanded to include active capability. *POSTGRES* (Stonebraker [25]) is an example of an integrated architecture.

The relative merits of the first option were discussed in section 1. A layered architecture has the advantage that it is easy to reuse passive technology. It has the disadvantage that coupling modes may be limited and external events can't be easily supported. The integrated architecture is the most versatile and flexible of the three, but it will require the most work to design, build and debug.

There is a lot of diversity in the architectural structure of the current generation of active databases. Often the best choice is dictated by the particular application domain and the extent to which it relies on any existing system. Further experimentation, however, will hopefully illuminate additional strengths and weaknesses of specific models.

3 Trigger Definition

The most widely used model for specifying triggers in active databases is known as the Event-Condition-Action rule or E-C-A rule (Dayal [12]). Its principal advantage is that of generality. Each of the event/condition/action is a separate part and they may have totally different semantics.

The event describes when the rule should be triggered (e.g when the database is modified, whenever a particular query is executed, whenever a certain time of day occurs)

The condition determines if a particular state of the database is true. (e.g when the number of tuples in relation x is less than 40)

The action describes the operations to be performed if the condition evaluates to true.

So we have

on Event

if Condition

then Action

As an example, the following trigger sounds an alarm if a particular customer tries to obtain an account.

on **insert to** customer

if customer.name = 'Mr Shonky'

then signal(alarm)

This is the most general model. It is also possible to have event-action (omitting the condition part) and condition-action (omitting the event part) rules. If the condition part is omitted, then we assume it's always true. If the event is omitted, then the rule is executed when the condition becomes true. This may be for only the first time the condition becomes true, or whenever there is a transition from false to true. An interesting approach using these latter types of rules is presented in Zaniolo [31], which attempts to unify active and deductive semantics based on an X-Y stratification mechanism , similar to EMS stratification in Kemp [21]. We now consider the kind of semantics most appropriate for specifying each of these three entities.

3.1 Event Semantics

In order to specify event semantics, it is necessary to consider what types of events need to be detected. A useful classification is to distinguish between primitive events and composite events (Chakravarthy [10] and Gehani [15, 16]). Primitive events can be treated as atomic, detectable occurrences. Composite events are combinations of various primitive events. Types of primitive events are

- database read/write/updates
- external events
- temporal events (relative, absolute and periodic)
- transaction events

External events are ones which occur outside the database boundary and could be the result of an application program signaling the database (e.g such as a CASE tool). Absolute temporal events occur when a particular point in time is reached (e.g 12:34am) and relative temporal events occur relative to another event (e.g 12 minutes after event x occurs). Periodic events are similar, but they occur a certain number of times (e.g every 5 minutes after event x). Transaction events are points at the beginning/end/at commit phase/etc of a transaction.

Primitive events can have attributes which may be used for passing information onto the condition and/or action part of the rule (Dayal [12]). Examples are user id, transaction id, time of occurrence, relation accessed etc. These attributes can be saved over one or more event occurrences. This could be useful in the situation where an event occurs multiple times within a transaction, yet one wants to fire an integrity checking rule only once for all of them.

Some of the operators that can be used to form composite events are listed below. For more examples see Chakravarthy [10] and Gehani [15, 16].

- Disjunction($\vee$) : The disjunction of events e1 and e2 ($e1 \vee e2$) occurs when either e1 or e2 occurs or when both e1 and e2 occur
- Sequence (;) : The sequence of e1 and e2 (e1;e2) occurs when both e1 and e2 have occurred in that order. Other events may occur between e1 and e2.
- Conjunction($\wedge$): The conjunction of events e1 and e2 ($e1 \wedge e2$) occurs when both e1 and e2 have occurred in any order, with possibly other events in between.
- Negation (not) : The negation of an event e1 (not e1 [interval]) occurs when e1 has not occurred during the specified time interval. The interval may be a simple time interval [4am - 10pm] or an interval between two events (between e2 and e3)

Not all of these operators are suitable for use with any type of event. It is useful to define a sub class of events called definite events (Chakravarthy [10]). These are events which are guaranteed to occur. The larger the class of definite events is, the greater the efficiency of the system, since there won't be as much wasted time trying to detect events that may never happen.

Most implementations do not have an event algebra as rich as that described above. Rather, they settle for being able to detect just database read/-write and updates and have no operators. While this obviously limits the range of situations which can be detected, it makes for a simpler system design and greater overall efficiency. Depending upon event complexity, some type of temporal logic would be most suitable for specifying event semantics.

3.2 Condition Semantics

A condition is essentially a query over the database state and its environment. Thus its semantics can be the same as those used for the user query interface(SQL, Datalog etc). This may be complex, include things like aggregation operators, negation and may refer to old and new states corresponding to the state before the execution of an operation and the state created after its execution. Mechanisms such as transition tables may be used to facilitate this (Widom [30]). A condition can be interpreted as being satisfied in two different ways. The first is if it produces any tuples when executed. The second is if it produces tuples different to when it was last executed. If the database is a temporal one, then the condition may refer to time values.

3.3 Action Semantics

Three broad categories of action languages can be defined (Voort [27])

- Basic query language: This language is the same as that used for interfacing with the database system and includes read/write/update operations
- Augmented query language: This is a superset of the basic query language with additional commands to support interaction with the external environment. (e.g printing messages to the user, triggering a statistical tool to operate on the database)
- Complex manipulation language: This aims to provide maximum flexibility in the type of operations possible. Rules are able to call arbitrary procedures. Possible actions are gathering statistics on database usage, restructuring of the database indexing scheme and automatically adjusting query optimisation strategies.

The difference between a complex manipulation language and an augmented query language is potentially very great, since the former can call arbitrary applications. Consequently, it may become impossible to reason about database behavior in a meaningful manner when these actions are executed.

4 Execution Model of Triggers

The execution model is a specification of the run time behaviour of the system. In particular, we want to describe when the various components of a rule are executed with respect to one another. A way of doing this is by defining coupling modes for each rule as done in *HiPAC* (Chakravarthy [11]). One mode for the event-condition part and another one for the condition-action part.

Possible coupling modes for event-condition are

- Immediate: The condition is evaluated immediately when the event becomes true within the current transaction. There is the choice of suspending the current operation and evaluating the condition, or completing the current operation and then evaluating the condition.
- Deferred: The condition is evaluated within the same transaction, but after the last operation in the transaction and just before it commits. This and the immediate mode can cause long delays in a transaction
- Decoupled/Detached: The condition is evaluated within a separate transaction.

The possible coupling modes of condition-actions are the same. Different types of coupling modes may be more/less suitable for certain categories of trigger, since the time of execution may have a significant effect on its outcome. Decoupled execution will help response time for normal queries, maintaining views should be done immediately and either deferred or immediate for integrity constraints (immediate for constraints that should never be violated and deferred for constraints that only need to be satisfied when the database is in a stable state.)

For decoupled execution, the problem arises of maintaining consistency between the initial transaction and the transaction it spawns (Etzion [13]). If the original transaction aborts, does this mean the sub-transaction should too ? Ways of approaching this include

- Make the the two transactions completely independent (useful for a logging situation where one wants an access log written regardless of whether its parent transaction committed or not)
- Have the sub-transaction abort if the parent one does
- Assign priorities to each transaction and have the sub transaction abort if its priority is lower than that of the parent
- Have the two transactions dependent for an initial period and then make them independent subsequently

It is this first option that makes decoupled execution very useful in situations where the original transaction is aborted because of some minor fault, yet it is crucial that a rule be triggered.

If a rule generates an error while executing, e.g due to data deletion, no access privileges or deadlock, a choice has to be made as to how to continue. Some choices are - abort the transaction, skip the current rule, skip all rules, re-process all rules.

Decoupled execution increases concurrency since it helps make transactions smaller. The number of locks taken up is also reduced. This concurrency could have the effect of violating causal independence though, since the condition transaction could theoretically see the results of the event transaction. Similarly, if the action of one rule triggers another rule, can this rule run before the action is completed ? (this is a possibility for action sequences which are complex)

Decoupled execution could well result in long delays between separate components of a rule being executed far apart. Some actions though, may be time critical and so it may be necessary to specify a limit in which it must be completed, otherwise the (sub)transaction should be aborted.

A correctness criterion for transactions involving rules is serializability which can be taken care of by the transaction manager.

A decision has to be made when a rule component is not executed immediately, as to what environment it should run under. Should it inherit the original environment, or is the current one more appropriate ? Is it useful to inherit the same coupling mode ?

For detecting complex events, it is necessary to maintain a structure called an event history (Gehani [16]). This records all events which have occurred and information about them. Since this can potentially be very large, some limit will have to be chosen as to how many events are 'remembered'. When temporal events are being detected, the event 'granularity' becomes very important. Are they signaled using seconds, minutes or hours. Event detection presents problems when events can affect each other. What will happen if the triggering event of a rule is invalidated by some other event before the rule has been considered, must the rule still be considered ? (this could occur if one rule triggers on the deletion of a particular tuple and then that tuple is reinserted before the condition is evaluated)

When bindings are passed between the condition and action part of a rule, a choice arises as to how many times the action should be executed. The action can be applied once for each data item in the binding set of the condition (known as instance or tuple oriented processing) or the action can be executed once for the whole binding set (known as set oriented processing).

5 Ordering of Triggers

If several triggers are activated, one needs to specify the order in which they are to be handled. This can happen in situations where several triggers share the same event rule. The number of possibilities can become quite large. For two rules a and b, the options are

- a and b fire in parallel
- a fires then b fires
- b fires then a fires
- a and b fire in random order
- only one of a and b fire

Obviously this grows very rapidly when the number of activated rules is large.

So, for a number of rules, the possibilities are

- fire them all in parallel. While this may maximise concurrency, it greatly raises the possibility of unpredictable behaviour. This is used in *HiPAC*.
- fire them all sequentially in a predetermined order. This necessitates some absolute/relative rule ordering scheme. Ties can be broken at random.
- fire them all sequentially in random order
- separate the rules into modules and fire the rule with highest priority in each module. This can be useful since it can limit conflicts to within a single module (Ceri [5]).
- fire only one rule chosen according to a priority scheme or at random.

Some possible priority schemes are

- Absolute priority: Each rule is assigned a numerical value and the rules are ordered according to its magnitude. Ties can be broken at random. This is used by *Ariel*.
- Relative priority: Each rule has an identifier signifying which rule(s) it has higher priority than. This is used in *Starburst*.
- Function based priority: One applies a function to each rule and the rules are ranked according to its value. This function could be state dependent or might be static considering only things like arity size, number of predicates etc.

Priority schemes can make optimisation of execution more difficult, though they can introduce more determinism. It will be necessary for the user to specify a priority for each rule, or else rely on default priorities.

6 Optimisation of Triggers

Since rules in an active database are likely to be evaluated many times, optimisation is very important. It is concerned with different ways of efficiently detecting, evaluating and executing triggers. Ways of optimising event detection are

- Petri net (Gatziu [14]) or finite state automaton (Gehani [15]). These may eliminate the need for a complete event history to be kept since their current state provides sufficient information about previous event occurrences.

- Introduce new types of locks into the lock table. Used in *POSTGRES*.

Brute force is the default way of evaluating conditions. Some possible optimisation strategies are

- Discrimination network such as Gator/RETE/-TREAT (Hanson [17, 18]). These are based on algorithms that were designed to optimise OPS5 rule programs in a main memory environment. They use an incremental method to maintain state information from one cycle to the next.

- Query rewrite. Rewrite the condition query such that it is less expensive to evaluate

- Tuple marking (Stonebraker [24]). Each rule is processed against the database and every record satisfying the event qualification is identified. Each such record is marked with a flag identifying the rule to be triggered. Therefore, every record is marked with zero or more flags indicating the rules to trigger if certain events operate on this record. This can require more storage, but no searching needs to be done at runtime. It's most economical when the majority of rules refer to only a few records. The main problem to deal with is keeping marking correct when updates occur.

- indexing and caching of rules (Antunes [2] and Stonebraker [24])

A way of optimising action execution is

- lazy/eager evaluation. Eager corresponds to executing actions as soon as possible. This may be useful for integrity constraints where an early transaction abort can be useful. Lazy corresponds to only executing the action when the result is needed. This is appropriate when the database being updated would otherwise be updated by the trigger more often than required. Used by *POSTGRES*.

7 Termination and Confluence of Triggers

When multiple triggers are defined in an active system, there is the potential for interference amongst them. Two important properties need to be looked at in this context, termination and confluence. A rule set is said to terminate if, for any database state and any initial triggering event, rules cannot activate each other indefinitely. A rule set is confluent if, for any database state and triggering event, the final database state is independent of the order in which the activated rules were executed.

Termination is a obviously a vital property to have, yet it's undecidable in the general case to determine whether it exists for an arbitrary rule set. Most previous work has considered termination in the context of specific rule systems. Some possible approaches are

- put restrictions on the rule groups (Zhou [32]).

- construct a triggering graph and if there are no cycles in it, then termination is assured (Aiken [1])

- term rewriting system. Express E-C-A rules in a term rewriting system and then use standard techniques for analysing termination and confluence (Karadimce [20]).

- use a propagation algorithm to determine whether one rule can affect another and then construct a triggering graph (Baralis [3])

Since it is difficult to analyze the termination properties of a rule set, it may be easier to put a limit on the possible triggering depth allowed, though correct executions could exceed this limit. It is undesirable to have rules looping to a large depth, since they will inhibit concurrency by keeping locks on data and possibly overload memory buffers.

Guaranteeing confluence may well be quite difficult, so one might settle for partial confluence, i.e rules are confluent for only some tables in the database. Confluence is guaranteed if all the rules commute (Baralis [3]), which is a very conservative property and not very useful in practice.

It would be most useful for the DBMS administrator to have a static analyzer of the rules set, which would give an indication of whether termination or confluence is possible. Also, where a problem existed, it would identify which rule(s) were responsible. This way the administrator could predict the likely outcome of adding a new rule to the rule set.

8 Management of Triggers

Management of a collection of triggers presents a number of problems. The first question deals with who is allowed to add/delete triggers. Should it be a privileged task for the database administrator, or can users do it as well ? Also, what are the access rights of triggers, will triggers defined by one user affect another user ? The answers to these questions will obviously depend on the environment, but it should be noted that due to their complexity, it is probably unwise to allow novice users to define their own triggers. This is because indiscriminate additions could well degrade system performance and have undesired side effects. The potential for malicious damage is also increased.

While it may be undesirable to allow everyone to manipulate the trigger collection, it is useful to let triggers activate/deactivate each other. The action language may have a facility for enabling/-disabling a particular trigger. Alternatively, a trigger may have some mechanism for ageing automatically and expire after a certain number of activations.

It would be useful for there to be a tool whereby users can find out what triggers are in existence. It would be possible to pose queries about rules and see which rules refer to which tables etc. Another useful feature is to include a dependency tracking system for rules. This ensures that when a condition or action can no longer be executed (e.g because a table has been deleted) then the rule should be invalidated. A related problem is trying to ensure that when new triggers are inserted, that they are consistent with the state of the database and consistent with the other rules present.

9 Existing Systems

We'll examine four active database systems

- *Ariel* (Hanson [17]): Developed at Wright Research and Development Centre. It focuses on constructing an active system which is efficient, reliable and carefully integrated with traditional DBMS functions.

- *POSTGRES* (Stonebraker [25]): Developed at the University of California, Berkeley as a successor to Ingres. It's a relational system with trigger functionality which is primarily meant as an implementation language for both materialised and non materialised views, partial views, procedures and caching of procedures.

- *Starburst* (Widom [30]): Developed at IBM Almaden. It's a relational system with a focus on extensibility so that it can serve as a testbed for innovations in database technology.

- *HiPAC* (Chakravarthy [11]): High Performance Active Database System, developed at Xerox. Its emphasis is on handling time constrained management problems. It uses an object oriented data model.

9.1 Ariel

Triggers are event/condition action rules, though the event may be omitted from the rule. Events are insertions/deletions/updates on a particular table. Conditions are predicates over the state of the database. The action is a sequence of insert/-delete/update commands on the database. Each trigger is given a priority (a number between -1000 and 1000). Both old and new values resulting from updates can be used in the condition/action. Tuple oriented processing is used and all parts execute in the same transaction. Event and condition optimisation uses the RETE algorithm.

9.2 POSTGRES

Triggers are event/condition/action rules. Events are triggered on the usual database operations and additionally can trigger on both time and date. Disjunction can be used to support complex events. The coupling mode is immediate. Trigger priority can be specified by a numerical value. Optimisation strategies include query rewrite, rule locks, indexing, caching and lazy/eager evaluation.

9.3 Starburst

Triggers are event/condition/action rules. Events are triggered on database operations and event disjunction is supported. The coupling mode is deferred. Uses a conflict resolution strategy similar to that of expert systems for rule ordering

9.4 HiPAC

Triggers are event/condition/action rules. Events are triggered on database operations, relative and absolute temporal events and external events. Disjunction, sequence and closure are the event operators. All three types of coupling mode are supported. Optimisation of events is by signal graphs and optimisation of conditions is done using an incremental approach.

9.5 Commercial Systems

Active technology is now beginning to gain acceptance in the commercial world.

Sybase [26] has some limited trigger functionality. They are event-action rules with the events being insert/delete/update. Only one trigger can be associated with an operation on a table. The action part is limited to a sequence of *SQL* statements. Immediate coupling mode is used. Triggering is limited to one level (i.e triggered actions do not themselves cause triggers to be fired).

The *SQL* standards 2 and 3 have mechanisms to support triggers. Triggers are Event-Condition-Action rules. Events are insert/delete/update on a single base table. The condition is an *SQL* predicate and the action is a sequence *SQL* statements. The condition/action are part of the same transaction as the event. Triggering is limited to one level.

10 Future Directions

We have seen that active databases are a powerful, yet complex, extension to traditional database technology. Although useful research systems have been built, there remains much work to be done before full active functionality becomes a commercial reality.

Performance metrics for active databases are needed so that different systems and the optimisation techniques they use can be meaningfully compared. Currently there are a bewildering array of options to choose from in design, with a number of divergent paths being followed. These choices include semantic structure of rules, their execution model and the overall architecture in which they're incorporated. As further research is undertaken, a consensus will hopefully emerge on standard features to be included in an active database.

Interaction between triggers and specifically their termination and confluence is another pressing issue. Complex systems are not attractive if they will result in non termination. Confluence is more dispensable, however, since confluence in transaction processing is not guaranteed even in today's generation of relational databases. Techniques for static analysis of these properties are a vital tool for the person in charge of administering the rule base.

Finally, there are areas which have had little investigation, yet offer much potential. These include distributed rule processing of triggers, the coupling of an active database and an expert system shell and automatic translation of higher level specifications to triggers. In conjunction with this, techniques for debugging active databases are a necessity.

Despite the challenges presented, the rewards offered by active functionality are great and as the field matures, it should prove an enduring force within the area of database technology.

References

[1] A. Aiken, J. Widom and J. M. Hellerstein. Behavior of database production rules: Termination, confluence and observable determinism. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, San Diego, June 1992

[2] F. Antunes et al. A pragmatic approach for integrating data management and task management: modeling and implementation issues. In *Proceedings of the International Conference on Extending Database Technology*, Venice, Italy, March 1990

[3] E. Baralis and J. Widom. An algebraic approach to rule analysis in expert database systems. To appear in *Proceedings of the 20th VLDB Conference*, 1994

[4] M. Carey, R. Jauhari and M. Livny. On transaction boundaries in active databases: a performance perspective. Tech. report, University of Wisconsin, No. 796, Nov. 1988.

[5] S. Ceri. A declarative approach to active databases. In *Proceedings of the Eighth International Conference on Data Engineering*, Phoenix, February 1992

[6] S. Ceri, P. Fraternali, S. Paraboschi and L. Tanca. Constraint enforcement through production rules: Putting active databases to work. In *IEEE Data Engineering*, 15(1-4):10-14, Dec. 92

[7] S. Ceri and J. Widom. Deriving incremental production rules for deductive data. *Technical Report IBM RJ 9071*

[8] S. Ceri and J. Widom. Managing Semantic Heterogeneity with Production Rules and Persistent Queues. In *Proceedings of the 19th VLDB Conference*, Dublin, 1993

[9] S. Chakravarthy. Architectures and Monitoring Techniques for Active Databases: An Evaluation. *Technical Report UF-CIS-TR-92-041*

[10] S. Chakravarthy and D. Mishra. Snoop: An Expressive Event Specification Language for Active Databases. Tech. Report UF-CIS-TR-93-007, University of Florida 1993

[11] S. Chakravarthy et al. HiPAC: A research project in active, time-constrained database management (final report). Technical Report XAIT-89-02, Xerox Advanced Information Technology, Cambridge, Massachusetts.

[12] U. Dayal, A. Buchmann and D. McCarthy. Rules are Objects Too: A Knowledge Model for an Active, Object-Oriented Database System. In *Proceedings of the Second International Workshop on Object Oriented Database Systems*, September 1988

[13] O. Etzion. Flexible Consistency Modes for Active databases applications. In *Information Systems, Vol. 18, No. 6, pgs 391-404*, 1993

[14] S. Gatziu and K. Dittrich. Detecting Composite Events in Active Database Systems using Petri Nets. In *Proceedings of the 4th International Workshop on Research Issues in Data Engineering*, Houston, Texas, February 1994

[15] N. Gehani, H. V. Jagadish and O. Shmueli. Compose: A system for composite event specification and detection. Tech. Report AT&T, Dec. 1992

[16] N. Gehani, H. V. Jagadish and O. Shmueli. Event Specification in an Active Object-Oriented Database. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, San Diego, June 1992

[17] E. Hanson. Rule condition testing and action execution in Ariel. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, June 1992

[18] E. Hanson and M. Hasan. Gator: An optimized discrimination network for active database rule condition testing. Tech. Report TR93-036, University of Florida, 1993

[19] J. V. Harrison and S. W. Dietrich. Incremental View Maintenance. In *Proceedings of the 5th Australasian Database Conference*, Christchurch, 1993

[20] A. P. Karadimce and S. D. Urban. Conditional term rewriting as a formal basis for analysis of active database rules. In *Fourth International Workshop on Research Issues in Data Engineering*, Houston, Texas, February 1994

[21] D. B. Kemp and K. Ramamohanarao. Syntactically recognizable modularly stratified programs. In *Proceedings of the 5th Australasian Database Conference*, January 1994

[22] A. Segev and J. L. Zhao. Data Management for Large Rule Systems. In *Proceedings of the 17th VLDB Conference*, Barcelona, Sept. 1991

[23] U. Schreier et al. Alert: An architecture for transforming a passive DBMS into an active DBMS. In *Proceedings of the 17th International Conference on Very Large Data Bases*, Barcelona, Spain.

[24] M. Stonebraker et al. On rules, procedures, caching and views in database systems. In *Proceedings of the 1990 ACM SIGMOD Conference on Management of Data*, Atlantic City, May 1990.

[25] M. Stonebraker and G. Kemnitz. The POSTGRES next-generation database management system. In *Communications of the ACM*, 34(10):78-92.

[26] Sybase, Inc. Berkeley, CA 94710. *Transact-SQL User's Guide, Release 4.2*, May 1990

[27] M. H. van der Voort and M. L. Kersten. Facets of Database Triggers. Tech. Report. Centre voor Wiskunde en Informatica, April 1993.

[28] D. S. Warren. Database updates in pure prolog. In *Proceedings of the International Conference on Fifth generation Computer Systems*, pgs 244-253, 1985

[29] J. Widom. Deductive and Active Databases: Two paradigms or ends of a spectrum ? In *Proceedings of the First International Workshop on Rules in Database Systems*, Edinburgh, August, 1993

[30] J. Widom. The Starburst Rule System: Language Design, implementation and applications. In *IEEE Data Engineering Bulletin, Special Issue on Active Databases*, 15(4):15-18, December 1992.

[31] C. Zaniolo and R. Sadri. A simple model for active rules and their behavior in deductive databases. In *Proceedings of the Second ICLP workshop on deductive databases*, Santa Margherita, Italy, June 1994

[32] Y. Zhou and M. Hsu. A theory for rule triggering systems. In *Advances in Database Technology EDBT '90*, Berlin, March 1990