Rule Management and Evaluation: An Active DBMS Perspective†

Sharma Chakravarthy‡
Xerox Advanced Information Technology
4 Cambridge Center, Cambridge, MA 02142.
Email: sharma@xait.xerox.com

Abstract An *active database management system* is characterized by its ability to monitor and react to both database and nondatabase events in a timely and efficient manner. There have been several attempts at integrating this capability with conventional, passive DBMSs. In fact, this capability has been shown to be pivotal for supporting a variety of database functions in an elegant manner.

Providing active capability requires several conceptual as well as architectural extensions to a conventional DBMS. One such extension — perhaps a critical one— is the component that is responsible for the efficient management and evaluation of triggers, alerters, or rules in general. This paper analyzes the requirements of rule processing in an active DBMS. For concreteness, we describe the approach taken in HiPAC to address some of these requirements and present an architecture for RuMES (Rule Management and Evaluation Subsystem) in terms of its functional components and its interface with other HiPAC components.

1. Introduction

Traditionally DBMSs have been passive: that is queries or transactions are executed only when explicitly requested. Many applications, such as inventory control, cooperative processing, and factory automation are not well served by passive DBMSs. These applications require automatic monitoring of conditions defined over the database state and a capability to take actions when the state of the underlying database changes. For example, in an inventory application the quantity of items available need to be monitored to make sure they do not fall below a threshold in which case they have to be reordered. Other real-time applications, such as process control, threat assessment, and battle management require timely response to critical situations in addition to automatic monitoring. For example, in a process control application the boiler pressure need to be monitored and if it increases by an undesirable amount, remedial measures need to be taken within a specified time to release the pressure and if that is not possible, then a contingency plan (e.g., sounding an alarm) may have to be executed. Again, passive DBMSs are not well suited for these applications. A common requirement of the above classes of applications (requiring active and time-constrained processing, respectively) is that the DBMS monitor conditions defined on the state of the database, and then, evaluate the conditions when the state of the database changes, to invoke specific actions (possibly subject to timing constraints).

In the absence of *explicit* support for monitoring application requirements, they have been realized using special purpose mechanisms in at least two ways: polling (querying the database periodically) and embedding/encoding condition detection and action invocation as part of transactions/applications. Neither of these two approaches is completely satisfactory. Polling requires that the periodicity with which the database is queried be tuned very precisely to obtain a timely response. Frequent polling will lead to thrashing (overloading the database with queries that return empty answers most of the time) whereas infrequent polling runs the risk of missing the response window. Embedding situation monitoring in application code severely compromises modularity and in addition limits the extent to which condition evaluation can be optimized.

An active DBMS incorporates efficient condition monitoring as an integral part by extending its functionality. It has been pointed out [STON82, STON85, DAYA88a] that such a capability is, in fact, useful for elegantly supporting a host of database functions including, integrity control, access control,

†This work was supported by the Defense Advanced Research Projects Agency and by Rome Air Development Center under contract No. F30602-87-C-0029. The views and conclusions contained in this paper are those of the authors and do not necessarily represent the official policies of the Defense Advanced Research Projects Agency, the Rome Air Development Center, or the U.S. Government.

‡This paper draws upon the research done as part of the HiPAC project in general and the situation monitoring task in particular. Besides the author, Barbara Blaustein and Arnon Rosenthal were co-responsible for the situation monitoring task. Other members of the HiPAC project team are: Alex Buchmann, Umeshwar Dayal (currently with DEC), David Goldhirsch, Meichun Hsu, Rivka Ladin, Dennis McCarthy, Richard McKee of XAIT, and Michael Carey, Miron Livny, and Rajeev Jauhari of the University of Wisconsin, Madison.

maintenance of: derived data, materialized views, and snapshots, and rule-based inferencing.

The remainder of this paper is structured as follows. Section 2 analyzes the requirements of rule processing in an active DBMS and in the process identifies several applicable techniques. Section 3 summarizes the salient features of the approach taken in HiPAC. Section 4 briefly describes the architecture of HiPAC, the rule processor, and interactions of the rule processor with the rest of the HiPAC components. Section 5 contains conclusions.

2. Requirements

Briefly, an active database management system monitors *conditions* triggered by *events* representing database updates or occurrences external to the database and if the condition evaluates to true then the *action* is executed. In the literature, alerters, triggers, daemons, active objects, and situation-action rules have been used for modelling active (DBMS) constructs. Rules, event-condition-action (or ECA) rules, and situation-action rules are used interchangeably in this paper.

Below we analyze the functional and system design requirements of an active DBMS in general and the role of RuMES (Rule Management and Evaluation Subsystem) in particular. In fact, it is useful to extend the concept of RuMES beyond its role as a component of a DBMS. The subsystem need to be viewed as a software component to be either integrated tightly with a DBMS or coupled with a heterogeneous array of components. Design of such a system entails a good understanding of the functional as well as system design requirements.

2.1. Functional Requirements

Efficiency: The set of all rules is likely to form a potentially large set of predefined queries that need to be efficiently managed and evaluated when specified events occur. The ability of the rule processor to evaluate complex rules efficiently is critical to the performance of an active database management system, since rule evaluation imposes an overhead on (possibly) every database update or other primitive event. Conventional query processing techniques are not adequate for optimizing rules as explained below necessitating extensions to existing ones as well as the development of new ones.

The characteristics of rules and their execution semantics present several new opportunities for optimization. First, rules are temporally persistent. That is, they have a longer life-span and as a result are likely to be evaluated many times. This suggests that several rules can be optimized simultaneously in a group requiring the use of techniques developed for multiple query optimization [FINK82, CHAK82, CHAK86, SELL88, ROSE88]. The effect of multiple query optimization can be further enhanced by materializing intermediate results judiciously (e.g., common subexpressions). Second, rules used for real-time applications are likely to have priorities or timing requirements associated with their execution. Optimization of such rules requires different techniques, such as exhaustive optimization, new buffering strategies, and the use of main memory processing strategies. Finally, application semantics as well as response time considerations require that rules be executed in different *coupling modes* (see [HSU88] for details) with respect to a triggering transaction: in *immediate mode* -- in-line expansion of the triggering transaction, in *deferred mode* -- prior to the commit of the triggering transaction, and in *detached mode* -- as a separate transaction. In addition, semantics of the *nested* execution of rules (rules triggered within the execution of another rule) need to be cleanly merged with the transaction semantics.

Also, the way in which events, conditions, and actions are grouped for execution with respect to their triggering transaction will also play a critical role in the selection of optimization techniques. If the entire rule is being optimized, subexpressions can be freely moved across the components during optimization. Furthermore, the knowledge of the actions and events can be beneficially used for optimizing the condition portion rather than doing it in isolation.

Expressiveness: Specification of events and conditions was not a primary consideration in the design of data manipulation languages for relational DBMSs. As a result, even the specification and enforcement of simple integrity constraints (e.g., domain constraints) is done in an ad hoc manner. Some of the earlier approaches [ESWA75, ESWA76] specifically addressed the enforcement of integrity constraints by extending the data model for specifying triggers.

In contrast, active DBMSs require enhancements to the data model in at least two ways: i) for specifying events and ii) for specifying conditions and actions. In addition to events that generally correspond to database operations, such as insert, delete, and modify, other types of events (e.g., temporal events, nondatabase events, and constructs for defining complex events) need to be supported.

Specification of conditions and actions also require extensions to the language supported by the data model. It is conceptually difficult to express conditions that refer to changes in the state of a database (e.g., transition constraints) using the query language. For example, it is difficult to express the condition "Retrieve all the High_Salaried_Employees whose salary increased by at least 10 percent", where High_Salaried_Employees is defined as employees who earn more than 35K.

Management of rules: Although efficient rule evaluation is important for supporting active DBMSs, an equally important aspect is the support for the management of rules. First, the ability to manipulate rules (add/delete/modify) as any other data object in the system is essential. Second, mechanisms for enabling and disabling individual rules or sets of them is often needed. For example, the set of rules activated while an aircraft is taxiing need to be disabled when it becomes airborne, and a different set of rules for the current context need to be activated. Finally, selective enabling and disabling of rules is also important. For example, once a specific item in the inventory has been placed on order, it may be necessary to deactivate the rule that evaluates the threshold condition, until the order is filled (lest the rule will be executed at every update of that item causing multiple orders to be written for the same item).

The management aspect influences the representation and the algorithms used for its manipulation. Enabling and disabling as well as add and delete operations should preferably be supported through incremental manipulations of the data structure used for storing the optimized versions in order to avoid frequent reoptimization.

A generic facility for supporting other DBMS functions: It has been shown [STON82, STON85, DAYA88a] that several DBMS functions can be elegantly supported by the active DBMS abstraction that were being realized by special purpose techniques. Constraint management including integrity and security enforcement, maintenance of derived data (e.g., views), and rule-based inferencing are a few examples. The model should not only be powerful enough to accommodate above functions but also provide mechanisms/constructs for their optimization. For example, for maintaining materialized views, modifications to the underlying relations from different transaction can be accumulated and then applied whereas the semantics of integrity enforcement may not offer such flexibility. Ability to invoke rules from another rule (nested/cascading rule execution) also seems to be important for supporting other functions (e.g., inferencing).

2.2. System Design Requirements

In addition to the functional requirements listed above, system design requirements need to be examined to ensure that the resulting subsystem can be interfaced with systems other than DBMSs. Below, we elaborate on two such requirements:

Well-defined interfaces: Defining the boundaries of the subsystem responsible for managing and evaluating rules, its interaction with other components of the system, and the specification of the information that flows between the components is especially important for a component that is intended to interface to a variety of systems. Unlike a query optimizer, this component cannot optimize rules in isolation, but needs to interact constantly with other components of a DBMS (e.g., transaction manager, object manager). Even the subsystem itself need to be designed modularly to avoid duplication (e.g., a transformation module may be shared both by the query optimizer and the rule optimizer).

Extensibility: From a modelling viewpoint, RuMES can be regarded as a mechanism to monitor abstract events which are specified in terms of primitive events using a specification language. Essentially, a rule processor extends the class of events detected and as a consequence it should be possible to *bootstrap* rule processors to extend the capabilities provided by an existing system. An object-oriented approach which hides the implementation details exporting only the interface specification seems to be pertinent for supporting extensibility.

3. HiPAC's Approach

In this section, we summarize HiPAC research results that are relevant to the specification and processing of rules. The following subsections indicate the approach taken in HiPAC for addressing some of the functional and system design requirements discussed in the previous section. For example, optimization subsection addresses the efficiency aspects; ECA rules, event algebra, and algebraic extensions address the expressiveness of rules; and signal graphs and rules as objects address the management issues.

Rule objects for specification and manipulation: In HiPAC, events, conditions, and actions are explicitly

recognized and are encapsulated into a rule. In addition, rules can have attributes and carry other context information (e.g., coupling mode). Events of rules are expressions in the event algebra. Condition is a query in the extended relational algebra. Actions are arbitrary expressions/programs (including new event definitions).

HiPAC rules serve as a single mechanism for specifying different kinds of entities (alerters, triggers, etc.) used for supporting active DBMSs. Rules permit declarative specification allowing the optimizer to choose an appropriate evaluation scheme. The separation of event, condition, and action components has several advantages: i) provides a unique interpretation of the rule (e.g., permits asymmetric events-condition specification), ii) permits the coupling of rules with triggering transaction in several ways; for example, event-condition, condition-action, and event-condition-action grouping are possible. iii) knowledge of the action portion of the rule (e.g., no side effects, action is a user-defined event specification) as well as the event portion of the rule (e.g., the event can be modified to unify with another event without changing the semantics) can be used for optimizing the rule. iv) perform transformations by pushing computations from action to condition and from condition to event with out changing the execution semantics (of course, taking the coupling mode into account), and v) two or more rules can be combined into a larger rule as part of optimization (intuitively, two rules can be combined into one rule if the action portion of one rule unifies the event portion of the other rule).

To facilitate management, in HiPAC, rules are treated as first class objects. There is a rule object class, and every rule is an instance of this class. A special operation, *fire*, is by default defined on the rule object class. Other special operations such as activate and deactivate are provided on the rule object class.

The main advantages to treating rules as first class objects are: i) rules can be related to other objects and can also have attributes. This is a convenient mechanism for grouping rules by context (e.g., all rules having "Airborne" as the value of the attribute *mode*), thus reducing the scope of rule searches. It is also possible to create subclasses of the rule object class and define special attributes or operations on these classes. ii) rules can be created, modified, or deleted in the same way that other object are. Also, they are subject to the same transaction semantics as other data objects are: a transaction must obtain a *read lock on a rule object in order to fire it*, and a write lock in order to modify, delete, or disable it; hence, a rule that is in the process of being fired by one transaction cannot be modified, deleted, or disabled by another.

Event algebra: An event algebra has been developed to support several types of events. They can be broadly classified into: primitive events and composite events. Primitive events are associated with data manipulation operations (e.g., insert, delete, or modify), clock time and abstract (nondatabase) operations. The abstract event are not be directly detected by a DBMS. These events and their arguments are defined in the model, but are detected and signalled by other systems or applications. For example the event Flight-Airborne (Flight#, destination, takeoff_time, aircraft), is signalled by an application when a flight takes off and results in the activation of a set of rules. Since primitive operations are not instantaneous, two events corresponding to the *beginning* and the *end* of an operation are recognized.

A simple rule that orders an item when the inventory falls below the threshold value for that item s shown below:

```
Event:      Update quantity_on_hand(item)
Condition:  quantity_on_hand(item) <
                    threshold(item)
Action:     submit_order(item)
```

Figure 3.1

Temporal events can be either absolute points in time, defined by a system clock (e.g., 8:00:00 a.m., Jan 1st 1989), relative (30 secs after event A occurred), or periodic (every day at 10:00:00 p.m.). Composite events are recursively defined using primitive events and the following constructors are provided for that purpose: disjunction, denoted (E1 | E2), is signalled when either E1 or E2 is signalled; sequence, denoted (E1; E2), is signalled when E2 is signalled, provided that E1 has been signalled before during the same transaction or rule firing; and closure (of E), denoted E*. For the sake of definiteness, a closure event need to be followed by another event, e.g., (E*;E') signalling the end of the closure. Details of event algebra and the computation of associated information can be found in [DAYA88b]. Figure 3.1 illustrates a simple ECA rule.

Event detectors and Signals: Detection of events is a basic requirement for monitoring. HiPAC supports

the notion of an *event detector* which is responsible for detecting a pre-defined class of events. For instance, a database event detector recognizes events that correspond to database operations. Similarly, a clock event detector recognizes the absolute or relative time of the day using the system clock. Usually, there is a set of values (parameters) that is associated with an event (e.g., for a delete operation, the attribute values of the tuple(s) that are deleted). For events that are not instantaneous (e.g., closure event) it is necessary to *accumulate* the parameters associated with the event (or even different events) over an interval (e.g., from the beginning to the end of a transaction) and treat the resulting set as the parameter associated with the event.

A *signal* is a message that describes the occurrence of an event and includes a *signal relation* containing the parameters for the event and other descriptive information about the event (e.g., transaction_id, time of occurrence). A *signal manager or dispatcher* is responsible for detecting composite events that require accumulation of parameters. Note that an event detector is a primitive signal manager which does not have to accumulate parameters. The signal abstraction is the only abstraction used in HiPAC to specify the interface between signal managers/event detectors. The same abstraction is useful for supporting abstract events also. As long as the signal is defined, it does not matter as to who detects the event or who packages the parameters that are relevant to an event. It may some times be possible to derive the signal relation specification from the description of an event (e.g., an update operation as an event will result in the signal consisting of old and new values) and some times it may be explicitly specified (e.g., Flight_airborne as an abstract event along with the signal object specification). It is also possible to optimize the amount of data passed in the signal relation using the knowledge of the condition and action. For example, if the condition of a rule with a closure event (on modify operation until the end of the transaction) does not use deleted portion of the tuples, then the size of the resulting signal relation can be reduced by passing only added values.

Algebraic extensions: The relational algebra used for expressing queries and manipulation of data does not elegantly support a fundamental concept — *changes* — that seems to be important for expressing conditions involving different database states. In HiPAC, a powerful operator **Changes** (and other auxiliary operators for its support) has been introduced to express conditions involving changes to stored as well as derived objects (e.g., views). The objective was to enhance the expressiveness of rules and in addition recognize and optimize expressions involving the operator introduced. Changes are usually expressed with respect to the state of the database before and after one or more intervening events.

To support the notion of changes and to provide a uniform means of capturing the semantics of changes to relations by database updates, the concept of a *Δrelation* is introduced. Informally, a Δrelation can be viewed as a wide relation (whose scheme is the juxtaposition of the schemes of the same relation with attribute names pre- and post-fixed with ˜) consisting of *old* and *new* values of tuples that were involved in the updates. Nulls are used to denote attribute values that did not exist earlier (e.g., for deletes and inserts). Figure 3.2 shows an Employee relation (EMP), a transaction T and the resulting Δrelation. A Δrelation accumulates *net* changes to a single relation, which is manipulated like any other relation by relational operators.

```
EMP      tid      Name     Sal

         @123     Joe       30k
         @456     Lynn      40k
         @321     Ann       28k

 T: {Insert   (@789   Ed   180   25k);
     Modify(@123, new_tuple =
                (@123, Joe, 222, 33k));
     Delete (@456)}

The  ΔRelation (called ΔEMP) is:

    Removals˜(ΔEMP)          Additions˜(ΔEMP)
  ------------------       ------------------
 |                  |     |                  |
 ˜tid    ˜Name   ˜Sal      tid˜    Name˜   Sal˜

 @123     Joe     30k      @123     Joe     33k
  -        -       -       @789     Ed      25k
 @456     Lynn    40k       -       -       -
```

Figure 3.2

As shown in Figure 3.2, operators **Removals˜** and **Additions˜** yield relations that are vertical partitions of a Δrelation.

Essentially, the operator **Changes** applied to a base relation, computes a Δrelation. For example, informally, **Changes**(EMP, ...) computes the

Δrelation ΔEMP. Let R and R' represent the old and new state of a relation, respectively. R' can be computed using R and ΔR. Also, by definition Changes(R, [R, ΔR]) = ΔR. Changes can also be applied to any arbitrary relational expression resulting in a Δrelation whose schema is based on the schema of the relational expression and whose contents are defined by Outerjoin (Δdifference(E, E')) where E is the relation obtained by evaluating the relational expression on the database state before the updates, E' is the relation obtained by evaluating the relational expression on the database state after the updates and Δdifference is the symmetric difference of the relations E and E' (that is, <(E - E'), (E' - E)>) with attribute names suitably modified. Figure 3.3 illustrates a rule whose condition is expressed in terms of the Changes operator.

```
Event:      update(EMP)
Condition:  C1: Select[
                    Changes(HSE, [HSE, ΔHSE]),
                    sal~ - ~sal > 0.1*~sal]
Action:      Dislay  tuples in C1.

where HSE (High_Salaried_Employees) is a
view defined as Select(EMP, sal > 35K)
```

Figure 3.3

Signal Graphs: A signal graph is a directed acyclic graph. There are two types of nodes, *data* nodes and *operator* nodes represented by rectangles and ovals, respectively. A leaf node represents either a signal relation or a stored relation. A leaf representing a signal relation is said to be active. Paths upward from an active leaf are active. All other paths are passive. Passive nodes and edges are shown in bold font.

An event/condition portion of a rule (or even an entire rule) is translated into a signal graph. Optimizing transformations take signal graphs as inputs and produce signal graphs as outputs. Merging (or grouping) of rules and combining of rules end-to-end correspond to merging of signal graphs. The set of all rules managed by the rule processor is a forest of signal graphs. The unit of execution scheduled by the scheduler (or the transaction manager) is also a signal graph.

The notion of a signal graph provides a single, elegant formalism accommodating: flow of signal objects, optimizing transformations (e.g., transformation of the Changes operator), efficient evaluation (using a combination of top-down and bottom-up evaluation as explained in the next subsection) and finally incremental manipulations (to support management of rules). Figure 3.4 shows the signal graph for the rule shown in Figure 3.4.

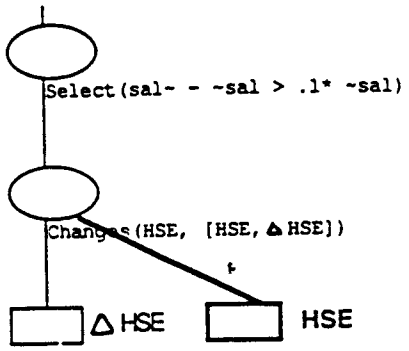

Figure 3.4. Initial signal graph

Optimization of Rules: In HiPAC, we identified several techniques as being especially useful for evaluating a set of rules efficiently: i) multiple condition optimization, ii) materialization of intermediate results, iii) optimization of events/signals, and iv) techniques for incremental optimization. In this paper, we only sketch the optimization of the Changes operator. Techniques for optimizing multiple queries can be found in [ROSE88]. Details of the optimization of Changes and other related optimizations are in [ROSE89, CHAK89].

Evaluation of Changes (or an expression involving Changes) from its definition using the old and the new relations requires computations involving R and R' and hence tend to be computationally expensive. Two transformations have been developed for optimizing the evaluation of Changes: an *incremental operator* to transform the evaluation of Changes on a single operator and a *chain rule* for transforming evaluation of Changes on an arbitrary expression.

Informally, an incremental operator (IncrOp) for a given operator (Op) is one that computes Changes(Op) without computing the new relation (R'). In other words, an incremental version of an operator computes Changes using the Δrelations (that are likely to be smaller) only and sometimes using the old relation in addition to the Δrelation. Incremental operators have been defined for all the relational operators. The definition of the incremental operator for Select is as follows:

```
IncrSel(ΔR, pred) =
     Outerjoin[Select(Removals~(ΔR), ~pred),
```

```
Select(Aditions⁻(ΔR), pred⁻),
⁻tid = tid⁻]
```

See [ROSE89, CHAK89] for the definition of incremental versions of other relational operators. Note that incremental versions can be defined for any operator, not just relational operators.

A chain rule is used to transform **Changes** to an arbitrary relational expression into an expression having incremental operators. Informally, the chain rule transformation pushes the **Changes** operator down the computation tree (replacing the nodes above the **Changes** by incremental versions, where possible) until it disappears from the tree. Figure 3.5 shows the optimized signal graph for the rule shown in Figure 3.3.

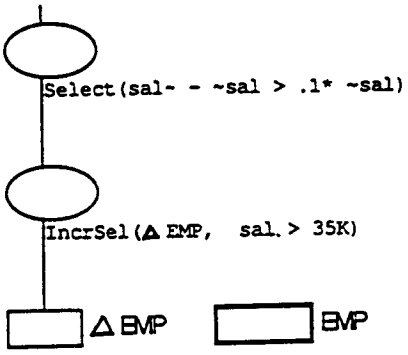

Figure 3.5 Optimized signal graph

Evaluation of a signal graph can also be optimized using the active and passive path distinction. Neither pure *bottom-up* nor *top-down* evaluation scheme is suited for the evaluation of signal graphs. Pure bottom-up evaluation evaluates each node before looking at its parents, e.g., in a postorder traversal. This results in unnecessary evaluation of nodes with passive subtrees. On the other hand, Top-down evaluation has each operator node (starting from the top) request input from its descendents. In this scheme all nodes between the outputs and the signal relations are examined, even when a signal relation is ∅. A combination of the two schemes -- bottom-up evaluation for active nodes interspersed with top-down evaluation for passive nodes provides an efficient evaluation scheme for signal graphs.

4. Architecture

In this section we present the functional architectures of HiPAC and RuMES. A detailed description of HiPAC's components and interfaces is beyond the scope of this paper. However, we briefly describe the components of HiPAC, present the components of RuMES, and trace the flow of a rule through the HiPAC system. Figure 4.1 shows the functional components of HiPAC. The object manager provides persistent storage for database objects. It also includes an event detector for primitive database events. The transaction manager is responsible for executing transactions, scheduling of subtasks, and includes an extended nested transaction model for supporting rule processing in HiPAC. Event detectors signal event occurrences through the signal message. RuMES is responsible for the management, processing including optimization and evaluation of HiPAC rules.

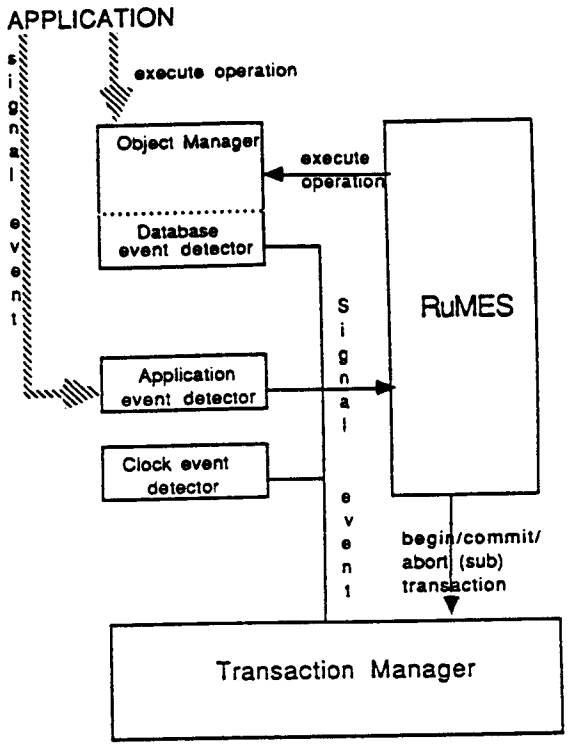

Figure 4.1. HiPAC Architecture

4.1. RuMES Architecture

The components of RuMES are shown in Figure 4.2. It consists of: a signal manager/dispatcher, a rule manager, and a rule processor composed of an optimizer and an evaluator. The signal manager receives signals ({<event-id + signal relations + descriptive data>}) as they asynchronously arrive from event detectors and other signal managers, accumulates them into one or more signal relations, and submits the resulting signal relation to the rule manager. The signal manager is primarily responsible

for detecting composite events defined by the event algebra [DAYA88b] and generating signal relations for them. The rule manager then examines signal relations to determine which rules need to be evaluated. The rule manager is, in general, responsible for the management of rules. This includes initial processing of rules, grouping them for optimization, maintaining the correspondence between rules and their optimized data structures, and supporting rule manipulation (addition, deletion, enable, and disable).

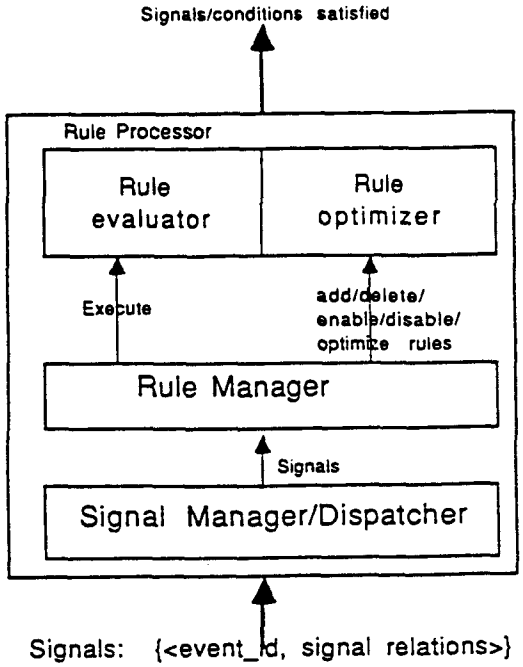

Figure 4.2. RuMES Functional Components

The rule processor is the analog of a query processor whose task is to apply optimizing transforms to signal graphs. The algorithms used for the efficient evaluation and incremental manipulation of signal graphs are also part of the rule processor.

We trace through the compilation, optimization, triggering, and execution of a rule below:

1. A rule specified to HiPAC, after some preprocessing, is transformed into an initial signal graph by the rule manager. During the transformation, events, condition, and actions are modified in a way similar to view modification. Events to be detected by the signal manager and the events that are detected outside of RuMES are determined. The rule is sent to the object manager for storage. The signal graph is finally passed to the optimizer.
2. The optimizer performs the transformations and indicates to the signal manager (or records in the data dictionary) the signal to be generated for that rule.
3. The signal manager of RuMES communicates the events to be detected to outside event detectors/signal managers. This is required for turning event detectors on/off for specific events based on disable/enable operations.
4. An event occurs and is signalled by an event detector to the signal manager. If the event is a database operation, then the operation is suspended.
5. The rule manager determines which rules are fired by the event and using the coupling mode information invokes the transaction manager to create necessary (sub)transactions and passes the rules to be evaluated. For each rule, the rule manager obtains the data that must be passed to condition evaluation and action execution.
6. The rule evaluator determines which of the rules are satisfied and returns a list. Again using the coupling mode information, the rule manager decides which actions need to be invoked and calls on the transaction manager to create concurrent subtransactions for each of these. Then the rule manager calls on the object manager to execute each rule's action in its subtransaction. If the action corresponds to an event, the signal is passed to the signal manager and the process repeats.
7. Delete/enable/disable operations are sent to the rule manager which chooses appropriate algorithms and creates transactions that are scheduled and executed like any other transaction by the transaction manager. Note that since rules (and consequently signal graphs) are first class objects, atomicity and persistence of the execution of these operations are guaranteed.

5. Conclusions

In this paper we have taken a very general-purpose approach to supporting rules in an active database management system. In contrast, others [STON87, DARN85, DITT86] have taken incremental approaches to supporting active DBMS functionality. [DITT86] has concentrated on consistency constraints in design databases. They do not address multiple and nested trigger execution as well as optimization of

their event/action triggers. [DARN87] supports events that correspond to database operations and has restrictions on the number of triggers that can be specified. In [STON87] rules are specified in QUEL (using query language syntax with special key words blurring the specification of events in the process) and supports clock events in addition to database events. Conventional query processor is used to optimize rules and persistent locks are used for event detection.

A breadboard has been implemented illustrating several functionality of the HiPAC system.

6. Acknowledgements

I would like to thank José Blakeley for his helpful comments on an earlier version of this paper.

7. References

[CHAK82] U. S. Chakravarthy and J. Minker. "Processing Multiple Queries in Database Systems." *Database Engineering*. 5. 3. pp. 38-44. 82.

[CHAK86] U. S. Chakravarthy and J. Minker. "Multiple Query Processing in Deductive Databases using Query Graphs". *Proc. of 12th VLDB Conf.*. Kyoto. Japan. August 86.

[CHAK89] Chakravarthy. U. S. et al.. "HiPAC: A Research Project in Active. Time-Constrained Database Management". Final Technical Report. Xerox AIT. 1989.

[DARN87] M. Darnovsky. J. Bowman. "TRANSACT-SQL USER'S GUIDE." Document 3231-2.1. Sybase Inc.. 1987.

[DAYA88a] U. Dayal. "Active Database management Systems". *Proc. of Conf. of Data and Knowledge Bases*. Jerusalem. 1988.

[DAYA88b] U. Dayal. A. Buchmann. D. McCarthy. "Rules are Objects Too: A Knowledge Model for an Active. Object-Oriented Database Management System." *Proc. 2nd Int'l Workshop on Object-Oriented Database Systems*. West Germany. Sept 1988.

[DITT86] K. R. Dittrich. A. M. Kotz. J. A. Mulle. "An Event/Trigger Mechanism to Enforce Complex Consistency Constraints in Design Databases." *SIGMOD Record 15*. No. 3. 1986. pp. 22-36.

[ESWA75] K. P. Eswaran and D. D. Chamberlain. "Functional Specifications of a Subsystem for Data Base Integrity". *Proc. of VLDB*. 1975.

[ESWA76] K. P. Eswaran. "Specifications. Implementations. and Interactions of a Trigger Subsystem in an Integrated Database System". IBM Research Report RJ1820. 1976.

[FINK82] S. Finkelstein. "Common Expression Analysis in Database Applications". *Proc. of ACM-SIGMOD*. Orlando. June 1982.

[HSU88] M. Hsu. R. Ladin. D. McCarthy. "An Execution Model for Active Database Management Systems". *Proc. 3rd International Conference on Data and Knowledge Bases*. June 88. pp. 171-179.

[ROSE88] A. Rosenthal and U. S. Chakravarthy. "Anatomy of a Modular Multiple Query Optimizer". *Proc. of VLDB*. Long Beach. 1988.

[ROSE89] A. Rosenthal. U. S. Chakravarthy. et al.. "Situation Monitoring in Active Databases". *Proc. of VLDB*. Amsterdam. 1989.

[SELL88] T. K. Sellis. "Multiple-Query Optimization". *ACM TODS*. Vol. 13. No. 1. 1988.

[STON82] M. Stonebraker. et al. "A Rules System for a Relational Data Base Management System." *Proc. 2nd Int'l Conf. on Databases*. Jerusalem. June 1982.

[STON85] M. Stonebraker. "Triggers and Inference In Database Systems." in *On Knowledge Base Management Systems* (Brodie and Mylopoulos.eds.) Springer-Verlag (1986).

[STON87] M. Stonebraker. M. Hanson. S. Potamianos. "A Rule manager for Relational database Systems." Technical Report. Dept. of Electrical Engineering and Computer Science. Univ. of California. Berkeley. 1987.