In: Won Kim, editor, *Modern Database Systems: The Object Model, Interoperability, and Beyond*, Addison-Wesley, Reading, Massachusetts, Sep. 1994

Active Database Systems

Umeshwar Dayal
Hewlett-Packard Laboratories
1501 Page Mill Road
Palo Alto, CA 94303
dayal@hplabs.hp.com

Eric N. Hanson
Dept. of Computer and Information Sciences
University of Florida
Gainesville, FL 32611
hanson@cis.ufl.edu

Jennifer Widom
Dept. of Computer Science
Stanford University
Stanford, CA 94305-2140
widom@cs.stanford.edu

Abstract

Integrating a production rules facility into a database system provides a uniform mechanism for a number of advanced database features including integrity constraint enforcement, derived data maintenance, triggers, alerters, protection, version control, and others. In addition, a database system with rule processing capabilities provides a useful platform for large and efficient knowledge-base and expert systems. Database systems with production rules are referred to as *active database systems*, and the field of active database systems has indeed been active. This chapter summarizes current work in active database systems; topics covered include active database rule models and languages, rule execution semantics, and implementation issues.

1 Introduction

Conventional database systems are *passive*: they only execute queries or transactions explicitly submitted by a user or an application program. For many applications, however, it is important to monitor situations of interest, and to trigger a timely response when the situations occur. For example, an inventory control system needs to monitor the quantity in stock of items in the inventory database, so that when the quantity in stock of some item falls below a threshold, a reordering activity may be initiated. This behavior could be implemented over a passive database system in one of two ways, neither of which is satisfactory. First, the semantics of condition checking could be embedded in every program that updates the inventory database, but this is a poor approach from the software engineering perspective. Alternatively, an application program can be written to poll the database periodically to check for relevant conditions. However, if the polling frequency is too high, this can be inefficient, and if the polling frequency is too low, conditions may not be detected in a timely manner.

An *active database system*, in contrast, is a database system that monitors situations of interest, and when they occur, triggers an appropriate response in a timely manner. The desired behavior is expressed in *production rules* (also called *event-condition-action rules*), which are defined and stored in the database. This has the benefit that the rules can be shared by many application programs, and the database system can optimize their implementation.

The production rule paradigm originated in the field of Artificial Intelligence (AI) with expert systems rule languages such as OPS5 [Brownston et al. 1985]. Typically, in AI systems, a production rule is of the form:

$$\textit{condition} \rightarrow \textit{action}$$

An inference engine cycles through all the rules in the system, *matching* the condition parts of the rules with data in working memory. Of all the rules that match (the *candidate set*), one is selected using some *conflict resolution policy*, and this selected rule is *fired*, that is, its action part is executed. The action part may modify the working memory, possibly according to the matched data, and the cycle continues until no more rules match.

This paradigm has been generalized to event-condition-action rules for active database systems. These are of the form:

on *event*
if *condition*
then *action*

This allows rules to be triggered by events such as database operations, by occurrences of database states, and by transitions between states (among other things), instead of being evaluated by an inference engine that cycles periodically through the rules. When the triggering event occurs, the condition is evaluated against the database; if the condition is satisfied, the action is executed. Rules are defined and stored in the database, and evaluated by the database system, subject to authorization, concurrency control, and recovery.

Such event-condition-action rules are a powerful and uniform mechanism for a number of useful database tasks: they can enforce integrity constraints, implement triggers and alerters, maintain derived data, enforce access constraints, implement version control policies, gather statistics for query optimization or database reorganization, and more [Eswaran 1976, Morgenstern 1983, M. Stonebraker 1982]. Previous support for these features, when present, provided little generality and used special-purpose mechanisms for each. In addition, the inference power of production rules makes active database systems a suitable platform for building large and efficient knowledge-base and expert systems.

While the power of active database systems was recognized some time ago, a true research field did not emerge until relatively recently [Dayal 1988]. However, the field has quickly blossomed, and it currently enjoys considerable activity and recognition. A number of powerful research prototypes have been built [Hanson 1992] [Chakravarthy et al. 1989] [Dayal et al. 1988] [McCarthy and Dayal 1989] [Gehani and Jagadish 1991] [Stonebraker et al. 1990] [Stonebraker and Kemnitz 1991] [Delcambre and Etheredge 1988] [Widom et al. 1991] [Widom and Finkelstein 1990] [Beeri and Milo 1991] [Cohen 1989] [Diaz et al. 1991] [Kotz et al. 1988] [Schreier et al. 1991] [Simon et al. 1992] [Buchmann 1990] [E. Anwar 1993] [S. Gatziu 1991]. In this chapter, we will illustrate the features of active database systems using *Ariel* [Hanson 1992], *HiPAC* [Chakravarthy et al. 1989, Dayal et al. 1988, McCarthy and Dayal 1989], *POSTGRES* [Stonebraker et al. 1990, Stonebraker and Kemnitz 1991], and *Starburst* [Widom et al. 1991, Widom and Finkelstein 1990] as representative of the field. Limited production rule capabilities are now appearing in commercial database products such as *Ingres* [INGRES 1992], *InterBase*, *Oracle* [ORACLE 1992], *Rdb* [Rdb 1991], and *Sybase* [Howe 1986], and in the emerging SQL2 and SQL3 standards.

This chapter provides a broad survey of current work in active database systems. The discussion is divided into three technical areas. Rule models and language design are discussed in Section 2, rule execution semantics in Section 3, and implementation issues in Section 4. Section 5 concludes and discusses areas for future research.

2 Rule Models and Languages

This section describes the issues involved in designing a database production rule language and explains how those issues have been addressed in various active database systems. We also describe the rule language features proposed for SQL2 and SQL3, which are indicative of the state of commercial practice.

Some of the differences among rule languages stem from differences in the underlying data models supported by the different systems. In relational systems such as Ariel, POSTGRES, Starburst, and the commercial products, rules are defined (and named) as metadata in the schema, together with tables, views, integrity constraints, and the like. As with other metadata, operations are provided to add, drop, or modify rules. In object-oriented systems such as HiPAC, rules are treated as first class objects that are instances of rule types defined in the schema. These rule types are subtypes of a generic type **rule**. Rules are structured objects, having events, conditions, and actions as their components. Like any object, rules can be created, deleted, or modified. In addition, rule objects have some special operations, including: **fire**, which causes a rule to be triggered; **enable**, which causes a rule to be activated; **disable**, which causes a rule to be deactivated (so that it won't be triggered even if its triggering event occurs).

Database rule languages vary considerably in the complexity of specifiable events, conditions, and actions. In some languages, the triggering event may be implicit—any relevant change to the database that can cause the condition to become true is treated as a triggering event. However, a key advantage of making events explicit is the flexibility gained in expressing transitions. For example, suppose it is desired to keep the salaries of two employees, Alice and Bob, the same. Suppose further that the following semantics are desired: if the constraint is violated because Alice's salary is changed by a user transaction, then change Bob's as well; however, if the constraint is violated because Bob's salary is changed by a user transaction, then abort the transaction. With explicit events, it becomes possible to specify these separate transitions.

In addition, some languages provide mechanisms whereby data (parameters) can be *bound* in the event and/or condition part of a rule, then passed to the condition and/or action. Some languages provide *rule ordering* as a conflict resolution mechanism. Finally, some languages provide mechanisms for organizing a large rule base. In the remainder of this section we address each of these issues in further detail.

2.1 Event Specification

The most common triggering events in active database rule languages are modifications to the data in the database. In relational database systems, these modifications take place through **insert**, **delete**, and **update** commands; in object-oriented database systems, these modifications may also take place through method invocations. All active database systems support rules that are explicitly or implicitly triggered by database modifications. In a relational database system, a rule explicitly triggered by database modifications might look like:

> **define rule** MonitorNewEmps
> **on insert to** employee
> **if** ...
> **then** ...

where *employee* is a table of employee information. In an object-oriented database system, a rule explicitly triggered by database modifications might look like:

> **define rule** CheckRaises
> **on** employee.salary-raise()
> **if** ...
> **then** ...

where *salary-raise* is a method defined over objects in an *employee* class.

Some rule languages also allow rules to be triggered by data retrieval:

> **define rule** MonitorSalAccess
> **on retrieve** salary
> **from** employee
> **if** ...
> **then** ...

Other database operations such as transaction **commit**, **abort**, or **prepare-to-commit** are allowed by some languages.

Some languages support rules triggered by *temporal events*. These might be absolute (e.g., *08:00:00 hours on 1 January 1994*), relative (e.g., *5 secs. after takeoff*), or periodic (e.g., *17:00:00 hours every Friday*).

Finally, a number of languages allow *composite events*, ranging from simple disjunctions of modification events to arbitrary combinations of events specified by powerful event composition operators.

The SQL2 standard allows *assertions* to be defined on tables. Each assertion is a simple rule that is triggered by one of the following events: **before commit**, **after insert**, **after delete**, or **after update** of a table. In the case of updates, a subset of the table's columns may be specified, so that the rule is triggered only when those columns are updated. The proposed SQL3 standard introduces **triggers** in addition to assertions (the difference will become clear when we discuss the action parts of these rules). The allowed triggering events are **before** or **after** an insertion, deletion, or update of a table.

The triggering events allowed in most relational research prototypes are similar to those for SQL2 assertions (although commit events typically are not allowed). Additionally, in Starburst, a rule may specify more than one modification operation on the same table; the rule is triggered when any of the operations occur (i.e., the event is a disjunction). In Ariel, the event may be omitted from a rule, in which case triggering is defined implicitly by the rule's condition. (Any event that causes the condition to be satisfied triggers the rule.) POSTGRES allows single explicit triggering events, which may be updates, some disjunctions of updates, or retrieval operations.

The object-oriented active database systems typically support a richer event specification language. In HiPAC, events can be generic database operations (**retrieve**, **insert**, **delete**, **update**), type-specific operations (method invocations) including operations on rule objects, transaction operations, temporal events, external events such as messages or signals from devices, and various compositions of these events, including disjunction, sequence, and repetition. Also, in HiPAC, events are defined to have formal parameters (e.g., *salary-raise(e:employee, oldsal:integer, newsal:integer)* has three parameters, *e* of type *employee*, and *oldsal* and *newsal* of type *integer*). When an instance of this event type occurs, the formal parameters are bound to a specific employee (the one whose salary is being updated) and two specific integers (this employee's old salary and new salary). Other object-oriented systems (e.g., DOM, Ode, Adam, Samos, and Sentinel) have proposed similar capabilities.

2.2 Condition Specification

In all database production rule languages, the condition part of a rule specifies a predicate or query over the data in the database. The condition is satisfied if the predicate is true or if the query returns a non-empty answer. When the event is explicit, the condition may often be omitted, in which case it is always satisfied. Many database rule languages allow conditions in rules triggered by database modifications to refer both to the modified data and to the database state preceding the triggering event. These mechanisms are described in Section 2.4. With such mechanisms, *transition conditions*, which are conditions over changes in the database state, may be expressed.

In the commercial systems, and in Ariel, POSTGRES, and Starburst, rule conditions are arbitrary predicates over the database state; modified data also can be referenced, so transition conditions can be specified. An example of such a rule is the following:

> **define rule** MonitorRaise
> **on update to** employee.salary
> **if** employee.salary $> 1.1 *$ **old** employee.salary
> **then** ...

In SQL2 assertions, the condition also is a predicate, but the condition is satisfied if the predicate is false. In HiPAC, rule conditions are sets of predicates or queries on the database; if all of the predicates are satisfied and all of the queries' results are non-empty, then the condition is satisfied. Transition conditions may be expressed in HiPAC using the event parameter mechanism described in Section 2.1.

2.3 Action Specification

The action part of a database production rule specifies the operations to be performed when the rule is triggered and its condition is satisfied. In AI rule languages, the action part of a rule usually inserts, deletes, or updates data in the working memory based on data matching the rule's condition. However, most database production rule languages allow more general rule actions.

In SQL2 assertions, the action part is implicit. When the condition is satisfied (i.e., when the predicate is false), the implicit action is to abort the current transaction.

In SQL3 triggers, Ariel, POSTGRES, and Starburst, rule actions can be arbitrary sequences of retrieval and modification commands over any data in the database. Rule actions also may specify **rollback** to abort the current transaction. All of these languages have a mechanism whereby rule actions can refer to the data whose modification caused the rule to be triggered (see Section 2.4).

Hence, if desired, rule actions can be based on triggering data as in AI rule languages. An example of this is the following:

define rule FavorNewEmps
on insert to employee
then delete employee e **where** e.name = employee.name

This rule is triggered whenever a new employee is inserted; its action deletes any existing employees with the same name. In POSTGRES, a rule's action may be tagged with the keyword **instead**, indicating that the action is to be executed instead of the triggering operation.

Rule actions in HiPAC can contain arbitrary database operations, transaction operations, rule operations, signals that user-defined events have occurred, or calls to application procedures.

2.4 Event-Condition-Action Binding

In AI rule languages such as OPS5, there is a link between the data that matches a rule's condition and the behavior of the rule's action. Each time an OPS5 rule is executed, the variables in the condition are bound to data items in working memory that satisfy the condition, and these are then passed to the action.

Since database production rule languages may have explicitly specified events, and since they have different and more varied conditions and actions than OPS5 rules, the notion of binding also is different and more varied. In HiPAC, the triggering event of a rule may be parameterized, and these parameters may be referenced in the rule's condition and action. For example, if a rule is triggered by *salary-raise(e:employee, oldsal:integer, newsal:integer)*, then *e* in the condition or action refers to the employee object on which the method was invoked, and *oldsal* and *newsal* refer to the integers bound when the event occurred. For composite events, the parameters have to be accumulated before being passed to the condition and action. For example, for the repetition event *salary-raise*(e:employee, oldsal:integer, newsal:integer)*, which allows the *salary-raise* event to occur one or more times, the data passed to the condition is a multiset of 3-tuples of *(e, oldsal, newsal)* values, one for each occurrence of the *salary-raise* event. Recall that rule conditions in HiPAC can be sets of queries. The results of these queries can also be referenced (along with the event parameters) in rule actions.

In SQL2/SQL3, Ariel, POSTGRES, and Starburst, rules are (explicitly or implicitly) triggered by insertions, deletions, and/or updates on a particular table. Hence, each rule language has a mechanism whereby the inserted, deleted, or updated tuples can be referenced in rule conditions and actions.

In SQL2/SQL3 and Ariel, when a rule is triggered by a change to a table T, then any reference to T in the rule condition or action implicitly references the changed tuple. This is illustrated by the following SQL3 trigger:

create trigger DeptDel
before delete on department
when department.budget < 100,000
delete employee **where** employee.dno = department.dno

This rule is triggered before the deletion of a department; the condition (**when** clause) and action refer to the department being deleted.

Old and new values of updated tuples can also be referenced in the condition and action parts, as illustrated by rule MonitorRaise in Section 2.2. SQL2 assertions use the keywords **old** and **new**. SQL3 triggers allow the trigger definition statement to give specific "correlation names" to the old and new values; these names are then used in the condition and action. In Ariel, the old value is referenced using the keyword **previous**. The entire table on which the change occurs, rather than just the changed tuples, can also be referenced in the condition and action parts. This is achieved through the use of *tuple variables* or synonyms as illustrated by the rule FavorNewEmps in Section 2.3.

In POSTGRES, the event-condition binding is similar: a reference to the table whose change triggered the rule implicitly references the changed tuple. However, in the action part, a reference to the table whose change triggered the rule produces the entire table. To reference the modified tuple before and after the triggering event, POSTGRES uses the special tuple variables **new** and **old**. For example, the *FavorNewEmps* rule in Section 2.3 would be written in POSTGRES as follows:

define rule FavorNewEmps
on append to employee
then delete employee **where** employee.name = **new**.name

In Starburst, a single rule triggering may involve arbitrary combinations of inserted, deleted, and updated tuples (as will be described in Section 3). These changes may be referenced in the condition and action part of a Starburst rule using *transition tables*. Transition tables are logical tables that are referenced just like database tables. At rule execution time, transition table **inserted** contains the tuples that were inserted to trigger the rule, transition table **deleted** contains the tuples that were deleted to trigger the rule, and transition tables **new-updated** and **old-updated** contain the new and old values, respectively, of the tuples that were updated to trigger the rule. As an example, the following Starburst rule aborts the transaction whenever the average of updated employee salaries exceeds 100:

> **define rule** AvgTooBig
> **on update to** employee.salary
> **if** (**select** avg(salary) **from new-updated**) $>$ 100
> **then rollback**

2.5 Rule Ordering

SQL2 and SQL3 do not allow more than one rule to be defined with the same triggering event, hence conflict resolution is never needed. However, most other rule languages do not impose such a severe syntactic restriction. In many active database systems, the choice of which rule to execute when more than one is triggered is made more or less arbitrarily, although some languages do provide features whereby the rule definer can influence conflict resolution.

Various features have been considered for POSTGRES, including *numeric priorities* and *exception hierarchies*, but none have been incorporated to date. In Starburst, rules are *partially ordered*. That is, for any two rules, one rule can be specified as having higher priority than the other rule, but an ordering is not required. In Ariel, rules have numeric priorities. Each rule is assigned a floating point number between -1000 and 1000; if no number is specified explicitly then a default of 0 is assigned. HiPAC departs from other active database systems in that multiple triggered rules are executed concurrently (see Section 3.5). Even so, HiPAC includes a mechanism whereby rules can be relatively ordered to influence the serialization order of concurrent execution.

2.6 Rule Organization

Since the number of rules defined in an active database system may be very large, some languages include mechanisms for organizing the collection of rules. Also, some include mechanisms for selectively activating and deactivating rules as a way of controlling the number of rules that must be monitored by the system at any given time.

In relational systems, rules are defined in the schema. Rules refer to particular tables, and so are subject to the same controls as other metadata objects (e.g., views, constraints); thus, if a table is dropped, all rules defined for it are no longer operative. No special mechanisms are provided for organizing the rules in a schema.

In HiPAC, rules are first class objects. Hence, they can be organized in types like any other objects. Rule types can participate in subtype hierarchies, they can have attributes, and they can be related to other objects. Like other objects, rules can be included in collections, which may be explicitly named or defined by queries. For example, one can define the collection

{ *r* **in** *Flight-rule* **where** *Effective-date(r) after 1/1/90* }

where *Flight-rule* is a rule type and *Effective-date* is an attribute defined for this type. Collections of rules can be selectively activated and deactivated using the **enable** and **disable** operations. These can be invoked from within a user transaction or in the actions of other rules.

3 Rule Execution Semantics

The semantics of a database production rule language determines how rule processing will take place at run-time once a set of rules has been defined, including how rules will interact with the arbitrary database operations and transactions that are submitted by users and application programs. Even for relatively small rule sets, rule behavior can be complex and unpredictable, so a precise execution semantics is very important.

As explained in Section 1, in AI rule languages rules are processed by an inference engine that cycles through all rules in the system, in each cycle finding those rules whose conditions are true and choosing one such rule whose action is then executed. In active database systems this approach is not always appropriate or adequate. Most importantly, unlike in AI systems, in active database systems rule processing is integrated with conventional database activity—queries, modifications, and transactions—and it is this activity that causes rules to become triggered and initiates rule processing. Furthermore, during rule processing in an active database system it may be too inefficient to determine, in each cycle, the entire set of rules whose conditions are true.

As with the rule language itself, there are a number of alternatives for rule execution, and there is considerable variance in the semantics taken by existing active database systems. One issue considered by active database system designers is the *granularity* of rule processing, especially for rules that are triggered by modification events. Since in most database system languages, operations work on sets of tuples (or sets of objects), there is a choice of firing the rule after each tuple (or object) is modified, or once for the entire set of modifications specified by the operation. Other choices are possible; for instance, rules may be fired at the end of an entire transaction.

Another issue is whether more than one rule can be triggered by the same event; some languages preclude this, and others allow it. If it is allowed, then is the execution sequential, using some form of conflict resolution to select one rule at a time, or can rules execute concurrently? Some languages have sequential execution semantics, while others allow concurrent execution. With either sequential or concurrent execution semantics, there is also the issue of whether one rule can trigger the execution of another rule or of (another instance of) the same rule. Clearly, if such nested triggering is allowed, termination is a concern.

Finally, the interplay between rule execution and the execution of user-initiated transactions is also an issue. Some systems execute rules within the scope of the triggering transaction, that is, the transaction in which the triggering event occurred. Others allow more flexible scoping of rules and transactions.

In this section, we separately consider the four example active database systems we have been discussing—Ariel, POSTGRES, Starburst, and HiPAC—and describe each system's approach to run-time rule execution. We also describe the semantics of rule execution in SQL2 and SQL3, and briefly consider the issue of error recovery during rule processing. For convenience, in the remainder of this section we use the term database "user" to mean user or application program.

3.1 Ariel

In Ariel, rules are triggered by *transitions*, which are database modifications induced either by a single database command or by a sequence of commands grouped together by the user to delineate rule processing. Since a single data modification command in relational systems such as Ariel is a set-oriented **insert**, **delete**, or **update** operation, the minimum *granularity* of rule processing in Ariel is a set of tuple-level operations. Commands grouped together may constitute an entire transaction, but they may not span transactions, so the maximum rule processing granularity is an entire transaction. Rule processing is invoked automatically at the end of each transition and takes place as part of the transaction containing the transition. Once rule processing begins, Ariel uses a cycling inference engine similar to that used by AI rule languages.

Recall from Section 2.4 that Ariel rule actions can reference the data whose modification triggered the rule. Since Ariel rules may be triggered by sets of changes, these references may correspond to sets of tuples rather than single tuples. As an example, consider the following rule:

define rule MonitorNewBobs
on insert to employee
if employee.name = "Bob"
then retrieve employee

If multiple tuples are inserted into the employee table before this rule is executed, then the rule's action will retrieve all of the inserted tuples whose value in column *name* is "Bob". In general, when a triggered rule is executed in Ariel, the rule processes the entire set of triggering changes, including both the user-generated changes that initiated rule processing and any subsequent changes made by rule actions. If a rule is executed multiple times during rule processing (e.g., because it is re-triggered by another rule's changes, or because it triggers itself), then each time it executes, it processes all matching changes since the last time it executed. If **rollback** is executed in a rule action, then rule processing terminates and the transaction is aborted.

Recall from Section 2.5 that each Ariel rule is assigned a numeric priority, but that the assignments need not be unique. Hence, when multiple rules are triggered, conflict resolution in Ariel proceeds as follows:

1. Pick the rule(s) with highest numeric priority.
2. If there's a tie, pick the rule(s) most recently matched by changes.
3. If there's still a tie, pick the rule(s) whose condition is the most selective.
4. If there's still a tie, pick a rule arbitrarily.

Finally, note that when Ariel rules are processed after a transition, the rules actually consider the *net effect* of the modifications in the transition, rather than the individual modifications. In most cases the net effect is the same as the individual modifications. However, in some cases there is a difference: if a tuple is updated several times in a transition, the net effect is a single update; if a tuple is updated and then deleted, the net effect is deletion of the original tuple; if a tuple is inserted and then updated, the net effect is insertion of the updated tuple; if a tuple is inserted and then deleted, the net effect is no modification at all.

3.2 POSTGRES

In POSTGRES rule processing is invoked immediately after any modification to any tuple that triggers and satisfies the condition of one or more rules. This sometimes is referred to as *tuple-oriented* rule processing, as opposed to Ariel's *set-oriented* rule processing. Recall that rule actions in POSTGRES are arbitrary database operations. Hence, when a rule's action is executed, it may modify multiple additional tuples, each of which may (immediately) trigger additional rules. Consequently, rule processing in POSTGRES is inherently recursive and synchronous (similar to a procedure call mechanism), rather than sequential as in Ariel. The basic rule processing algorithm in POSTGRES is described as follows:

1. Some (user- or rule-generated) tuple modification occurs.
2. The modification triggers and satisfies the conditions of rules R_1, R_2, ..., R_n.
3. For each rule R_i execute R_i's action.

As mentioned above, action execution (step 3) can perform tuple modifications that recursively invoke this rule processing algorithm. There is no conflict resolution mechanism in POSTGRES—triggered rules are executed in arbitrary order. If **rollback** is executed in a rule action, then rule processing terminates and the transaction is aborted.

As a simple example of the difference between tuple-oriented and set-oriented rule processing in relational systems, consider the following rule:

define rule SetSalary
on insert to employee
then begin
starting-salary := (**select** avg(salary) **from** employee) − 10 ;
update employee (salary = starting-salary) **where** employee.id = **new**.id
end

This rule is triggered by insertions into the employee table; its action sets the starting salary for inserted employees to 10 less than the average employee salary. Suppose a set of new employees is inserted. In a tuple-oriented rule system such as POSTGRES, this rule is triggered once for each inserted employee, so the salaries of the new employees differ. In a set-oriented rule system such as Ariel, this rule is triggered only once, so the salaries of the new employees are the same.

3.3 Starburst

In Starburst, rule processing is invoked automatically at the end of each user transaction that triggers one or more rules. In addition, users can invoke rule processing within transactions by issuing special commands. Hence, as in Ariel, the minimum rule processing granularity is a single relational database command (i.e., a set of tuple-level operations) and the maximum granularity is an entire transaction.

We first explain end-of-transaction rule processing in Starburst, then describe rule processing within transactions in response to user commands. Recall that Starburst rules may be triggered by inserts, deletes, and/or updates; a rule is triggered whenever one or more of its triggering operations occurs. During Starburst rule processing, the first time a triggered rule is executed it considers all modifications since the start of the transaction, including the user modifications and any subsequent modifications made by rules. If the rule is triggered additional times, it considers all modifications since the last time it was triggered. Like Ariel, Starburst rules consider the net effect of sets of modifications, rather than the individual modifications (recall Section 3.1).

Starburst rule processing uses an inference engine similar to AI rule languages and to Ariel. However, in each cycle Starburst determines the rules that are triggered, but does not eliminate those whose condition is false—a triggered rule's condition is not evaluated until the rule is selected for consideration. For conflict resolution, recall that Starburst rules may be assigned relative priorities. Hence, when a triggered rule is selected for condition evaluation and possible execution, it is selected such that no other triggered rule has higher priority.

In addition to automatic rule processing at the end of each transaction, rule processing in Starburst is invoked within transactions when the user issues one of three commands: **process rules**, **process ruleset** S, or **process rule** R. Command **process rules** invokes the same rule processing algorithm that is invoked at transaction end. Command **process ruleset** S also invokes rule processing, but only for those rules in the user-defined rule set S. Command **process rule** R is similar, except only rule R can be triggered or executed. Regardless of whether a rule is executed in response to one of these commands or in response to end-of-transaction rule processing, the semantics is the same: the rule considers the entire set of modifications since it was last considered within the transaction, or since the start of the transaction if it has not yet been considered. As in the other systems, if **rollback** is executed in a rule action, then rule processing terminates and the transaction is aborted.

3.4 SQL2 and SQL3

SQL2 and SQ3 permit both tuple-level and set-level processing of assertions and triggers. The choice is made at rule definition time by specifying a **for each row** option (which induces tuple-level processing) or omitting it. Recall that assertions can be defined with a database modification command as the triggering event or with the special event **before commit**. In either case, the condition is evaluated over the current state of the database, including all modifications made since the start of the transaction. Triggers are triggered by database modifications; they can refer to the states of the modified table prior to (**old**) or immediately following (**new**) the modification.

Rule processing is strictly sequential. No conflict resolution is necessary, since no two rules can be defined to have the same triggering event. Further, additional syntactic restrictions on rule definitions ensure that the same table cannot be modified multiple times in a sequence of rule firings, thereby ensuring termination. Note that none of the other systems guarantees termination.

3.5 HiPAC

Before describing run-time rule processing in HiPAC, it is necessary to introduce the concept of *coupling modes*. Coupling modes originated in the HiPAC project but subsequently have been discussed in the context of other active database systems, e.g. [Gehani and Jagadish 1991, Schreier et al. 1991, S. Gatziu 1991, E. Anwar 1993, Buchmann 1990]. Coupling modes determine how rule events, conditions, and actions relate to database transactions. Whereas in Ariel, POSTGRES, Starburst, and many other active database systems, rule conditions are evaluated and actions are executed in the same transaction as the triggering event, in HiPAC this is not always the case. The rule definer has the flexibility of deciding whether or not the conditions and actions should execute in the triggering transaction.

Let E, C, and A denote the event, condition, and action, respectively, of a rule. Associated with each HiPAC rule is an *E-C coupling mode* and a *C-A coupling mode*. The E-C coupling mode determines when the rule's condition is executed with respect to the triggering event, and the C-A coupling mode determines when the rule's action is executed with respect to the condition. Each coupling mode is either: *immediate*, indicating immediate execution; *deferred*, indicating execution at the end of the current transaction; *decoupled*, indicating execution in a separate transaction. Not all combinations of coupling modes make sense; Figure 1 shows the seven combinations that are allowed and the two that are not. For each of these combinations, it is relatively easy to construct an active database application for which that behavior seems most appropriate. In addition, for the decoupled mode, a *causality* constraint can optionally be specified; if specified, this constraint means that the triggered transaction can commit only if the triggering transaction commits, and the triggered transaction must follow the triggering transaction in the serialization ordering.

Rule processing in HiPAC is invoked whenever any event occurs that triggers one or more rules.

	C-A Mode		
E-C Mode	*immediate*	*deferred*	*decoupled*
immediate	condition checked and action executed after event	condition checked after event, action executed at end of transaction	condition checked after event, action executed in separate transaction
deferred	not allowed	condition checked and action executed at end of transaction	condition checked at end of transaction, action executed in separate transaction
decoupled	condition checked and action executed in separate transaction	not allowed	condition checked in one separate transaction, action executed in a different separate transaction

Figure 1: Coupling modes in HiPAC

As mentioned in Section 2.5, HiPAC differs considerably from most other active database systems in its handling of multiple triggered rules. Rather than selecting one triggered rule to execute using some form of conflict resolution, HiPAC executes all triggered rules concurrently. If, during rule execution, additional rules are triggered, they also are executed concurrently. To do this, HiPAC uses an extension of the *nested transaction* model of execution [Moss 1985], which lends itself well to this rule processing semantics and to the realization of coupling modes.

The basic rule processing algorithm in HiPAC is described as follows:

1. Some (user- or rule-generated) event triggers rules R_1, R_2, ..., R_n.
2. For each rule R_i schedule a transaction to
   a. evaluate R_i's condition;
   b. if the condition is true, schedule a transaction to execute R_i's action.

Transaction scheduling in step 2 is based on R_i's E-C coupling mode, while transaction scheduling in step 2b is based on R_i's C-A coupling mode: Immediate mode causes a nested sub-transaction to be spawned immediately, deferred mode causes a nested sub-transaction to be spawned at the commit point of the current transaction, and decoupled mode causes a separate (top-level) transaction to be spawned. Note that both condition evaluation and action execution (steps 2a and 2b) can generate events that recursively invoke this rule processing algorithm. Finally, as mentioned in Section 2.5, HiPAC rules may have relative ordering, and this ordering is used to influence the serialization order of concurrently executing nested sub-transactions.

3.6 Error Recovery

One issue not yet fully addressed in many active database systems is the semantics of error recovery during rule processing. A database rule may generate an error during its execution for a number of reasons—for example, because data it read has been deleted, because data access privileges have been revoked, because concurrently executing transactions have created a deadlock, because of a system-generated error, or because the rule action itself has uncovered an error condition.

Errors such as missing data or revoked privileges can usually be avoided in any database system with a sophisticated enough dependency-tracking facility. In such systems, when a data item is deleted or privileges are revoked, rules that depend on their existence are invalidated. Most database rule systems handle errors during rule processing by aborting the current transaction, since this is how conventional database systems typically handle errors during transaction processing. However, in the case of error conditions produced by rule actions, this is not the only possible reasonable behavior. Other alternatives are to terminate execution of that rule and continue rule processing, to return to the state preceding rule processing and resume database processing, or to restart rule processing.

The nested transaction model used in HiPAC allows some of these possibilities. When a rule execution subtransaction fails, the failure event is returned to its parent, which has the option of spawning a sibling subtransaction to repair the error (this may be accomplished through the firing of another rule that is triggered by the failure event). Alternatively, failure can be propagated up the transaction tree all the way to the root (top) transaction.

Another issue is how to recover events after system crashes. For events that are database operations, there is no problem: these are recovered as part of normal transaction recovery. For temporal or external events (such as those supported by HiPAC), events have to be declared to be recoverable or not; for recoverable events, their occurrences and parameter bindings have to be reliably logged. As part of recovery, uncommitted transactions (for the decoupled conditions and actions) triggered by the recovered event signals have to be restarted.

4 Implementation Issues

Active database systems must support all of the features provided by conventional database systems, including data definition, data manipulation, storage management, transaction management, concurrency control, and crash recovery. In addition, active database systems must provide mechanisms for event detection and rule triggering, for condition testing, for rule action execution, and for user development of rule applications.

4.1 Characteristics of Representative Systems

The Ariel active database system is built using the *Exodus* database toolkit [Carey et al. 1991]. The focus of Ariel's implementation is on efficient condition testing, which is achieved by incorporating a highly tuned *discrimination network* that extends the *Rete* and *TREAT* networks used by AI rule languages [Wang and Hanson 1992]. When data modification commands are executed in Ariel, the modified tuples are packaged as *tokens* and passed to the discrimination network, where rule conditions are tested. In addition, the Ariel architecture includes the following components:

- A *rule manager/rule catalog* for handling rule definition and manipulation tasks.
- A *rule execution monitor* for maintaining the set of triggered rules and scheduling their execution.
- A *rule action planner*, which is invoked by the rule execution monitor to produce optimized execution strategies for database commands occurring in rule actions; these commands are executed by the same query processor that executes user commands.

In POSTGRES, two different mechanisms are implemented for rules: *tuple level* processing and *query rewrite*. When a rule is created, the user selects which mechanism is to be used for that rule. Tuple level processing places a *marker* on each tuple for each rule that has a condition matching that tuple. When a tuple is modified or retrieved, if the tuple has one or more markers on it, then the rule or rules associated with the marker(s) are located and their actions are executed. Markers must be installed and removed when rules and data are created, deleted, and modified. In contrast, the query rewrite implementation consists of a module between the command parser and the query processor. This module intercepts each user command and augments it with additional commands reflecting the effects of rules triggered by the original command. Since the additional commands also may trigger rules, query rewrite must be applied recursively; in some cases it may not terminate. However, when applicable, the "compile-time" approach of query rewrite can be considerably more efficient than the "run-time" approach of tuple level processing. Unfortunately, the semantics can differ between the two approaches, as explained in [Stonebraker and Kemnitz 1991].

The Starburst database system has as one of its primary goals *extensibility* [Haas et al. 1990], and the rule system implementation relies on Starburst's extensibility features. The *attachment* feature is used to monitor data modifications that are of interest to rules. These modifications are stored in a main-memory data structure called a *transition log*. When rules are processed at the end of a transaction or in response to a user command, the transition log is consulted to determine which rules are triggered. Triggered rules are indexed in a sort structure reflecting rule priorities; rule conditions are evaluated and actions are executed through Starburst's normal query processor. References to transition tables (recall Section 2.4) are implemented using Starburst's *table function* feature: table functions for each of the four transition tables use the transition log to produce appropriate tuples at run time. The Starburst rule system also includes components for concurrency control, authorization, and crash recovery.

The HiPAC architecture extends an object-oriented database system with: a *rule manager*, which coordinates rule processing; *event detectors* for the different types of events. It also extends the functions of the *object manager* to store rule objects and implement operations on them; and the functions of the transaction manager to implement the coupling modes of the execution model, and to provide concurrency control and recovery for rule objects in addition to data objects. Algorithms for incremental evaluation of rule conditions after database modifications were also developed. There are three different main memory prototype implementations of HiPAC. The most substantial of these is a Smalltalk-80 implementation, which includes both a rule manager and a transaction manager. Concurrent transactions are implemented as Smalltalk *threads* (i.e., light-weight processes). A unique feature of this implementation is its support for bidirectional interaction between application programs and the database rule system: applications can invoke database operations, and rules running inside the database can invoke application operations.

4.2 Rule Programming Support

The implementation of an active database system can include many useful features that support the rule programmer. Features for analyzing rule processing include the ability to trace rule execution, to display the current set of triggered rules, to query and browse the set of rules, and to cross-reference rules and data. Other useful features include the ability to control errors in rule programs, to activate and deactivate selected rules or groups of rules while the database system is processing transactions, and to experiment with rules on an off-line subset of a working database. Simple versions of some of these features exist in some active database systems, while more sophisticated and complete versions will certainly emerge over time.

4.3 Rule Termination

Rule processing is subject to infinite loops, that is, rules may trigger one another indefinitely. In a database system this behavior can be catastrophic; for example, rules could erroneously fill the disk with data by repeatedly performing inserts on a table, eventually crashing the system. At the very least, a transaction in which rules are looping would surely inhibit concurrency (by holding locks on data) and saturate memory buffers, slowing system throughput. Recall that SQL2 and SQL3 avoid this problem by imposing sufficient syntactic restrictions on rule definitions; however, these restrictions are very strong and limit the expressiveness of the rule language. In general, given the power of the other rule languages discussed in this survey, it is an undecidable problem to determine in advance whether rules are guaranteed to terminate, although conservative algorithms have been proposed that warn the rule programmer when looping is possible [Aiken et al. 1992]. A run-time solution to detecting and preventing infinite loops is to provide a rule triggering limit. In this case, the number of rules executed during rule processing is monitored; if the limit is reached, rule processing is terminated. Most active database systems provide such a limit, specified by the user and/or by a system default. Another mechanism is to detect if the same rule is triggered a second time with the same set of parameters.

5 Conclusions and Future Directions

This chapter has provided an overview of active database systems, including database production rule models and languages, rule execution semantics, and implementation issues. The rules provided by active database systems can be used for integrity constraint enforcement, derived data maintenance, authorization checking, versioning, and many other database system applications; they also enable more advanced and powerful applications, and they provide a platform for large and efficient knowledge-base and expert systems.

The theory and technology of active database systems is still maturing. There are several areas that researchers and practitioners will likely address in the future, particularly as active databases emerge in the commercial arena. These include the following:

Support for application development: In Section 4.2 we described a number of features, not present in many active database system prototypes, that are vital for the development of database rule applications. One suggested approach to application development treats database rules as "assembly language", automatically generating rules from higher level specifications [Ceri 1992, Ceri and Widom 1990, Ceri and Widom 1991]. While this approach works well for a number of standard applications, there will always be a need to develop applications using rules directly. In addition, considerable work is needed on increasing the communication capability between database rules and applications.

Increasing the expressive power of rules: Some applications may need the ability to define rules with more complex triggering events, conditions, or actions than currently can be expressed in database rule languages. Methods for increasing the expressiveness of database rule language while maintaining an efficient implementation deserve further study.

Improved algorithms: Efficient algorithms for processing rules are crucial for delivering the functionality of active databases without excessively degrading the performance of conventional database processing. While some work has been done in this area, continued improvements are needed.

Distribution and Parallelism: So far, active databases have been considered primarily in centralized database environments. An initial consideration of the problem of rule processing in distributed and parallel environments appears in [Ceri and Widom 1992], but this is only a theoretical study relating to a particular rule language. Many issues arise when considering a distributed or parallel active database system, including the distribution and fragmentation of rules and algorithms that guarantee equivalence with centralized rule processing. Note that, even in centralized database systems, parallelism might be exploited to improve the performance of rule processing.

References

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

[Beeri and Milo 1991] Beeri, C. and Milo, T. (1991). A model for active object oriented database. In *Proceedings of the Seventeenth International Conference on Very Large Data Bases*.

[Brownston et al. 1985] Brownston, L., Farrell, R., Kant, E., and Martin, N. (1985). *Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming.* Addison-Wesley, Reading, Massachusetts.

[Buchmann 1990] Buchmann, A. (1990). Modelling heterogeneous systems as a space of active objects. In *Proceedings of the Fourth International Workshop on Persistent Object Bases.*

[Carey et al. 1991] Carey, M. et al. (1991). The architecture of the exodus extensible dbms. In Dittrich, K., Dayal, U., and Buchmann, A., editors, *Object-Oriented Database Systems.* Springer-Verlag, Berlin.

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

[Ceri and Widom 1990] Ceri, S. and Widom, J. (1990). Deriving production rules for constraint maintenance. In *Proceedings of the Sixteenth International Conference on Very Large Data Bases.*

[Ceri and Widom 1991] Ceri, S. and Widom, J. (1991). Deriving production rules for incremental view maintenance. In *Proceedings of the Seventeenth International Conference on Very Large Data Bases.*

[Ceri and Widom 1992] Ceri, S. and Widom, J. (1992). Production rules in parallel and distributed database environments. In *Proceedings of the Eighteenth International Conference on Very Large Data Bases.*

[Chakravarthy et al. 1989] Chakravarthy, S. et al. (1989). HiPAC: A research project in active, time-constrained database management (final report). Technical Report XAIT-89-02, Xerox Advanced Information Technology, Cambridge, Massachusetts.

[Cohen 1989] Cohen, D. (1989). Compiling complex database transition triggers. In *Proceedings of the ACM SIGMOD International Conference on Management of Data.*

[Dayal 1988] Dayal, U. (1988). Active database management systems. In *Proceedings of the Third International Conference on Data and Knowledge Bases.*

[Dayal et al. 1988] Dayal, U. et al. (1988). The HiPAC project: Combining active databases and timing constraints. *SIGMOD Record*, 17(1):51–70.

[Delcambre and Etheredge 1988] Delcambre, L. M. L. and Etheredge, J. N. (1988). The Relational Production Language: A production language for relational databases. In *Proceedings of the Second International Conference on Expert Database Systems.*

[Diaz et al. 1991] Diaz, O., Patom, N., and Gray, P. (1991). Rule management in object-oriented databases: A uniform approach. In *Proceedings of the Seventeenth International Conference on Very Large Data Bases.*

[E. Anwar 1993] E. Anwar, L. Maugis, S. C. (1993). A new perspective on rule support for object-oriented databases. In *Proceedings of the ACM SIGMOD International Conference on Mangement of Data.*

[Eswaran 1976] Eswaran, K. P. (1976). Specifications, implementations and interactions of a trigger subsystem in an integrated database system. Technical Report RJ 1820, IBM Research Laboratory, San Jose, California.

[Gehani and Jagadish 1991] Gehani, N. and Jagadish, H. V. (1991). Ode as an active database: Constraints and triggers. In *Proceedings of the Seventeenth International Conference on Very Large Data Bases*.

[Haas et al. 1990] Haas, L. et al. (1990). Starburst mid-flight: As the dust clears. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):143–160.

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

[Howe 1986] Howe, L. (1986). Sybase data integrity for on-line applications. Technical report, Sybase Inc.

[INGRES 1992] INGRES (1992). *INGRES/SQL Reference Manual,* Version 6.4. ASK Computer Co.

[Kotz et al. 1988] Kotz, A. M., Dittrich, K. R., and Mulle, J. A. (1988). Supporting semantic rules by a generalized event/trigger mechanism. In *Proceedings of the International Conference on Extending Data Base Technology*.

[M. Stonebraker 1982] M. Stonebraker, e. a. (1982). A rules system for a relational database management system. In *Proceedings of the Second International Conference on Databases*.

[McCarthy and Dayal 1989] McCarthy, D. R. and Dayal, U. (1989). The architecture of an active database management system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*.

[Morgenstern 1983] Morgenstern, M. (1983). Active databases as a paradigm for enhanced computing environments. In *Proceedings of the Ninth International Conference on Very Large Data Bases*.

[Moss 1985] Moss, E. (1985). *Nested Transactions: An Approach to Reliable Distributed Computing*. MIT Press, Cambridge, Massachusetts.

[ORACLE 1992] ORACLE (1992). *ORACLE7 Reference Manual*. ORACLE Corporaton.

[Rdb 1991] Rdb (1991). *Rdb/VMS – SQL Reference Manual*. Digital Equipment Corporation.

[S. Gatziu 1991] S. Gatziu, A. Geppert, K. D. (1991). Integrating active concepts into an object-oriented database systems. In *Proceedings of the Third International Workshop on Database Programming Languages*.

[Schreier et al. 1991] Schreier, U., Pirahesh, H., Agrawal, R., and Mohan, C. (1991). Alert: An architecture for transforming a passive DBMS into an active DBMS. In *Proceedings of the Seventeenth International Conference on Very Large Data Bases*.

[Simon et al. 1992] Simon, E., Kiernan, J., and de Maindreville, C. (1992). Implementing high-level active rules on top of relational databases. In *Proceedings of the Eighteenth International Conference on Very Large Data Bases*.

[Stonebraker et al. 1990] Stonebraker, M., Jhingran, A., Goh, J., and Potamianos, S. (1990). On rules, procedures, caching and views in data base systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*.

[Stonebraker and Kemnitz 1991] Stonebraker, M. and Kemnitz, G. (1991). The POSTGRES next-generation database management system. *Communications of the ACM*, 34(10):78–92.

[Wang and Hanson 1992] Wang, Y.-W. and Hanson, E. N. (1992). A performance comparison of the Rete and TREAT algorithms for testing database rule conditions. In *Proceedings of the Eighth International Conference on Data Engineering*.

[Widom et al. 1991] Widom, J., Cochrane, R. J., and Lindsay, B. G. (1991). Implementing set-oriented production rules as an extension to Starburst. In *Proceedings of the Seventeenth International Conference on Very Large Data Bases*.

[Widom and Finkelstein 1990] Widom, J. and Finkelstein, S. J. (1990). Set-oriented production rules in relational database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*.