In: The Knowledge Engineering Review, vol. 8 no. 2, pages 121–143, June 1993

An Overview of Production Rules in Database Systems

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

Jennifer Widom
IBM Almaden Research Center
650 Harry Road
San Jose, CA 95120 USA
widom@almaden.ibm.com

Abstract

Database researchers have recognized that 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, 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 paper summarizes current work in active database systems and suggests future research directions. Topics covered include database rule languages, rule processing semantics, and implementation issues.

1 Introduction

Database systems provide persistent storage for massive amounts of data and powerful interfaces for querying and modifying this data. Even so, most database systems are passive, since all data manipulation occurs in response to user or application commands. Database researchers have observed for some time that if a database system also provides mechanisms for creating and processing *production rules* (also referred to in this context as *triggers* or *alerters*), then the database system becomes active, and many useful capabilities can be provided by this active behavior [17, 34].

The production rule paradigm originated in the field of Artificial Intelligence (AI) with expert systems rule languages such as OPS5 [4]. While some work has been done on using database systems to provide efficient underlying support for OPS5 (e.g. [38, 42]), in this paper we focus on database systems with fully integrated production rules facilities, often referred to as *active database systems*. The AI production rule paradigm has been modified for the active database context so that rules can respond to database operations, occurrences of database states, and transitions between states, among other things.

Database researchers have discovered that with the addition of production rules facilities, database systems gain the power to perform a number of useful database tasks with one uniform mechanism: they can enforce integrity constraints, monitor data access and evolution, maintain derived data, enforce protection schemes, maintain version histories, and more. (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.

One could argue that the functions performed by database production rules could instead be performed by the database's application programs. In fact, application programs can be and are being used in this way. One approach is for an application program 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. A second approach is to include condition checking in every program that modifies the database, but such decentralization is a poor approach from the software engineering perspective. Integrating a production rules facility directly into the database system provides the same functionality as these application program approaches but does not suffer from their obvious disadvantages.

While the power of active database systems was recognized some time ago, a true research field did not emerge until relatively recently. However, the field has quickly blossomed, and it currently enjoys considerable activity and recognition. A number of powerful prototype systems have been built in the research context, while more limited production rule features are appearing in products and proposed standards. Important research projects, all of which are described in this paper to some extent, include (alphabetically) *Ariel* [24], *HiPAC* [9, 11, 32], *Ode* [19], *POSTGRES* [40, 41], *RPL* [13], and *Starburst* [45, 46]. A number of other research projects are not described in detail in this paper, either because the projects are fairly preliminary, because their features do not differ significantly from projects that are covered, or simply for lack of space; these are described in [3, 10, 15, 31, 37, 39]. In the commercial realm, some production rule capabilities are provided in *Ingres* [2], *InterBase*, *Oracle* [36], *Rdb* [16], and *Sybase* [28]. Finally, very simple rule capabilities have been incorporated into the SQL2 standard, while somewhat more general capabilities have been proposed for SQL3. Products and standards are not covered in this paper since their capabilities are subsumed by research projects that are covered.

There is a substantial body of work on another kind of database system with rules—deductive database systems. Deductive database systems are similar to conventional database systems in that they are passive, responding only to commands from users or applications. However, they extend conventional database systems by allowing the definition of PROLOG-like rules on the data and by providing a deductive inference engine for processing recursive queries using these rules. Deductive and active database rule systems are fundamentally different, and both types of rules could theoretically be present in a single system. We focus on active database systems and do not cover deductive database systems here. Readers interested in deductive database systems can consult the extensive literature on the subject, e.g. [6, 43].

This paper provides a broad survey of current work in active database systems, covering general issues and concepts as well as describing the several research projects mentioned above. (As a default, the Ariel system is used for illustrative purposes when appropriate.) The paper is organized as follows. Preliminary concepts and terminology necessary for non-database experts to understand the remainder of the paper are provided in Section 2. Technical coverage of active database systems is divided into three major areas: database rule language design in Section 3, rule processing semantics in Section 4, and implementation issues in Section 5. Finally, Section 6 contains concluding remarks, discussion of current and potential application areas, and predictions and recommendations for future research.

2 Preliminaries

Until recently, most database systems research has considered *relational* database systems, which now enjoy prominence among commercial vendors. However, some drawbacks of relational systems have led to the emergence of *object-oriented* database systems, which are prevalent among researchers and very recently have emerged as products. Some work in database production rules has focused on relational database systems, while other work has focused on object-oriented database systems. In particular, among the projects covered in this paper, Ariel, POSTGRES, RPL, and Starburst are relational, while HiPAC and Ode are object-oriented. Some active database concepts and issues are common to both types of database systems, while others are particular to one or the other, as will become evident in subsequent sections. As a default, we consider relational systems when the distinction is unimportant. In the remainder of this section we briefly introduce basic concepts and terminology for relational and object-oriented database systems. For further details see, e.g. [43].

In a relational database system, all data is stored in *tables* (or *relations*). Associated with each table is a fixed number of *columns* (or *attributes*). In a given state of the database, each table contains zero or more *tuples* (or *rows*), where each tuple assigns a value to each column of the table. Queries are posed against the database using a declarative database language such as *SQL* or *Quel*. A query can perform a variety of operations, including retrieving the entire contents of a table, retrieving data matching a particular predicate, and retrieving data in multiple tables joined by value comparison. Modifications to the database also use a query language such as SQL or Quel; data can be *inserted* (sometimes called *appended*), *deleted*, or *updated* (sometimes called *replaced*).

In an object-oriented database system, all data is stored as *objects*. Rather than adhering to a fixed format such as rows in tables, objects may have arbitrarily complex structure. Sets of objects with the same structure are grouped into *classes*. Typically, objects may refer to other objects through pointers (or *object identity*), rather than by comparing values as in the relational model. No prominent languages have emerged for querying object-oriented databases; some languages strive for similarity to declarative relational query languages, while others adopt a more "navigational" approach. In all languages, modifications to database objects are made through *method* invocations. A set of methods is associated with each class; methods are named procedures that operate on database objects in that class. Again, note that methods provide arbitrary operations rather than a fixed set of system-defined operations as in relational systems. Finally, classes can be arranged into *hierarchies*, so that classes inherit structure and methods from their ancestors.

In both relational and object-oriented database systems, database operations (queries and modifications) are issued either directly by users or, more commonly, by application programs. Operations typically are grouped into *transactions*. Transactions are executed by the database system in such a way that either all operations in a transaction are executed or, in the case of system failure or inconsistency, none of them are; i.e. transactions are *atomic*. Furthermore, if multiple users or applications operate on the same database at the same time, execution is guaranteed to appear as if each transaction is executed in isolation from other concurrent transactions, i.e. the transactions are *serializable*. When a transaction completes we say it has *committed*; when a transaction is interrupted (and its partial effects are undone) we say it has *aborted* or *rolled back*. Synchronization of simultaneous transactions is performed by the database system's *concurrency control* mechanism, and atomicity of transactions is guaranteed by the system's *crash recovery* facility.

3 Database Rule Languages

This section describes the many issues involved in designing a database production rule language and explains how those issues have been addressed in various active database system projects. The semantics of rule processing at run-time is then discussed in Section 4.

As mentioned in Section 1, database production rule languages have their roots in AI production rule languages such as OPS5. Generally speaking, AI production rules take the form:

$$\textit{pattern} \rightarrow \textit{action}$$

We call such rules *pattern-based*. The rule is *triggered* when the *pattern* is matched by data in the *working memory*, and the *action* modifies the working memory, possibly according to the matched data. (The pattern also may be referred to as a *condition* or *predicate*.) As mentioned in Section 1, some work has been done on using database systems to support such rule languages, but we focus here on the modification of these rule languages for fully integrated active database systems.

In most pattern-based rule languages, there is an implicit assumption that during rule processing a rule is triggered only when there is new data in the working memory that matches the pattern (or, in the case of *negated* rule conditions, when data matching the pattern is deleted from the working memory). Hence, rules implicitly are triggered by *events* such as the insertion, deletion or modification of data. The most obvious difference between AI rule languages and database rule languages is that in many database rule languages the triggering event or events are specified explicitly. Such rules take the form:

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

We call such rules *event-based.* The rule is triggered when the event occurs. Once the rule is triggered, the condition is checked on the data. (We call it a condition rather than a pattern since the notion of pattern-matching is less appropriate here.) If the condition is satisfied, the action is executed.

Most database production rule languages are event-based, although some are pattern-based and some support both forms. Database rule languages vary considerably in the complexity of specifiable events, conditions, and actions. In addition, some languages provide mechanisms whereby data can be *bound* in the event and/or condition part of a rule, then passed to the condition and/or action. Finally, some languages provide mechanisms for *rule ordering*, to determine which rule is executed when multiple rules are triggered. (This usually is referred to as *conflict resolution.*) In the remainder of this section we address each of these issues in further detail.

3.1 Event Specification

The most common triggering events in database production 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 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 database production rule languages also allow rules to be triggered by data retrieval. A different class of triggering events proposed in some systems is *timing events*. Rules with timing events might be triggered at particular times or time intervals. Finally, a number of database rule languages allow *composite events*, ranging from simple disjunctions of modification events to arbitrary combinations of data and timing events specified by powerful event languages.

We first survey the triggering events allowed in the relational active database system projects. In Ariel and Starburst, each rule is triggered by insertions, deletions, or updates on a particular 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. In Starburst, a rule may specify more than one triggering operation (on the same table); the rule is triggered when any of the operations occur, i.e. the events behave as a disjunction. In Ariel, the event may be omitted from a rule, in which case triggering is defined implicitly by the rule's condition (i.e. the rule is pattern-based), as will be described in Section 3.2. POSTGRES allows single explicit triggering events as in Ariel; in addition, rules in POSTGRES may be triggered by data retrieval operations. In RPL, rules are purely pattern-based, so no triggering events are specified.

We next consider the object-oriented systems. In Ode, rules are purely pattern-based. (Recent work in the context of the Ode project has suggested a rich event specification language [20], but this event language has not been integrated with Ode's rule language.) The HiPAC project allows by far the most complex triggering events of any database rule language, although it must be stated that the HiPAC language has not been implemented to the same extent as the other projects covered here. In HiPAC, rules can be triggered by *generic* database operations (**retrieve**, **insert**, **delete**, **update**), by method invocations, by transaction operations such as **commit** and **abort**, by temporal events (absolute, relative, and periodic), and by external events such as messages or sensor information. A rule also may be triggered by various compositions of these events, including disjunction, sequence, and repetition.

3.2 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. When the condition is a query, the meaning usually is that if the query produces any data then the condition is satisfied. In event-based rules, the condition usually may be omitted, in which case it is always satisfied. Many database rule languages have a mechanism whereby conditions in rules triggered by data modifications may refer to the modified data, or to the database state preceding the triggering modification. These mechanisms are described in Section 3.4. With such mechanisms, *transition conditions* may be expressed, which are conditions over changes in the database state.

We first consider conditions in purely event-based rule languages. In POSTGRES, rule conditions are arbitrary predicates over the database state, and transition conditions cannot be specified. In Starburst, rule conditions also are arbitrary predicates over the database state. Starburst has a mechanism for referencing modified data, so transition conditions may be specified. In HiPAC, rule conditions are sets of queries on the database; if all of the queries are non-empty then the condition is satisfied. Transition conditions may be expressed in HiPAC using a special mechanism for passing parameters (such as modified data) from a rule's triggering event to its condition.

In Ariel, a rule may be event-based, pattern-based, or both. A purely event-based rule specifies only a triggering event (and an action). A purely pattern-based rule specifies only a condition (and an action). Rule conditions in Ariel are arbitrary predicates over the database state, similar to POSTGRES and Starburst; transition conditions are expressible using a mechanism for referencing modified data. An example of a rule in Ariel that is both event- and pattern-based is:

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

This rule is triggered whenever a new employee tuple is inserted whose value in column *name* is “Bob”. Purely pattern-based rules in Ariel are triggered whenever new data in the database satisfies the rule’s condition (similar to expert systems rule languages such as OPS5). An example of a purely pattern-based rule in Ariel is:

> **define rule** MonitorNewBobs2
> **if** employee.name = “Bob”
> **then** ...

This rule is triggered whenever there is new data in the employee table whose value in column *name* is “Bob”, regardless of what data modification command created that data (i.e. insert or update). Finally, an example of a pattern-based rule in Ariel with a transition condition is:

> **define rule** RaiseLimit
> **if** employee.salary > 1.1 $*$ **previous** employee.salary
> **then** ...

This rule is triggered whenever there is new data in column *salary* whose value is at least 10% more than the previous value in that column.

In RPL, rules are purely pattern-based, and rule conditions are expressed as arbitrary queries over the database. If the query produces any data then the condition is satisfied. Similar to Ariel, it is implicitly understood that a rule is triggered only when new data “satisfies the condition”, i.e. new data is produced by evaluating the query. There is no mechanism for referencing modified data in RPL, so transition conditions cannot be expressed.

In Ode there are two types of rules, *constraint* rules and *trigger* rules. In both cases, each rule is associated with a particular class. (Recall that Ode is an object-oriented database.) In the case of constraint rules, each rule associated with a class is implicitly triggered by any invocation of any method on any object in that class. In the case of trigger rules, a rule remains dormant (i.e. it cannot be triggered) until one or more commands are issued to *activate* the rule. Each command to activate a trigger rule specifies a particular object and particular parameters; hence, a single trigger rule may be activated multiple times on multiple objects. Once activated on an object, a trigger rule is implicitly triggered by any invocation of any method on that object. In both constraint and trigger rules, the rule condition is a predicate over the value of the object on which the triggering method has been invoked. In constraint rules, the condition is true if the predicate is false; in trigger rules, the condition is true if the predicate is true, and the rule's activation parameters may be referenced in the condition. Transition conditions are not expressible in either case. Examples of constraint and trigger rules in Ode are given in Section 4.3

3.3 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 expert systems 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 pattern. This same approach is taken in RPL.[1] However, most database production rule languages allow more general rule actions. In 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. (Details on run-time rule processing are given in Section 4.) All three systems have a mechanism whereby rule actions can refer to the data whose modification caused the rule to be triggered (see Section 3.4). Hence, if desired, rule actions can be based on triggering data as in expert systems rule languages. An example of this is:

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

This (purely event-based) 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 change.

In Ode, rule actions are single statements, but since a statement can be a method invocation, this essentially allows arbitrary rule actions. With respect to rule actions, HiPAC again has the most generality (but again we observe that HiPAC has not been fully implemented). Rule actions in HiPAC can contain arbitrary database operations, signals that user-defined events have occurred, or calls to application procedures.

[1] To some extent, the RPL project falls into the class of database support for OPS5. However, we include RPL in this survey because it modifies OPS5 for the database setting and hence represents a bridge between expert systems and database rule languages.

3.4 Event-Condition-Action Binding

In expert systems rule languages such as OPS5, there is a link between the data that matches a rule's pattern and the behavior of the rule's action. Each time an OPS5 rule is executed (or *fired*), there is an *instantiation* associated with that execution: a data item, or combination of items, that matches the rule's pattern. At rule execution time, the values of the instantiated items can be referenced in the rule's action through the use of variables specified in the rule's pattern. That is, at run-time, variables are *bound* in the pattern and 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 expert systems rules, the notion of binding also is different and more varied. The most general approach once again is taken by HiPAC. 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, a rule may be triggered by *inserted*(i), where i in the condition or action then refers to the inserted value; a rule may be triggered by invocation of a method *GiveRaise*(o), where o in the condition or action then refers to the object on which the method is invoked; a rule may be triggered by a sensor signal *Detected*(x,y,z), where x, y, and z in the condition or action then refer to the values detected by the sensor. Recall that rule conditions in HiPAC are sets of queries. Through an additional mechanism, the results of these queries can be referenced (along with the event parameters) in rule actions. In Ode, there is no special mechanism for condition-action binding. Note, however, that since a rule in Ode is triggered by a method invocation on a particular object, the value of that object is available to the rule action.

In RPL, condition-action binding is similar to OPS5—rules are executed for particular instantiations of the rule condition (query), and these instantiations can be referenced in the rule action. In 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 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 rule, part of which appeared as an example in Section 3.2:

define rule MonitorNewBobs

on insert to employee
if employee.name = "Bob"
then retrieve (employee)

This rule is triggered whenever a new employee named "Bob" is inserted; its action retrieves the new tuple.

The binding feature in Ariel does not preclude rule conditions and actions from referencing the entire table on which the change occurs; this is achieved by using *tuple variables*. As an example, the following Ariel rule is triggered whenever an employee is deleted; its action raises the remaining employees' salaries by 10% of the deleted employee's salary:

define rule DistributeWealth
on delete to employee
then update employee e (e.salary = e.salary + .1 ∗ employee.salary)

Finally, when an Ariel rule is triggered by an update command, the keyword **previous** can be used in the rule condition or action to reference the old value of the updated tuple. A rule illustrating this was given in Section 3.2:

define rule RaiseLimit
if employee.salary > 1.1 ∗ **previous** employee.salary
then ...

In POSTGRES, the notion of event-condition-action binding is similar to Ariel but the syntax is somewhat different. In the condition part of a POSTGRES rule, a reference to the table whose change triggered the rule implicitly references the changed tuple, as in Ariel. However, in the action, a reference to the table whose change triggered the rule produces the entire table. To reference the changed value, POSTGRES uses keywords **new** and **old**. If a rule is triggered by inserts, then **new** references the inserted value and **old** is undefined. If a rule is triggered by deletes, then **old** references the deleted value and **new** is undefined. If the rule is triggered by updates, then **new** and **old** reference the new and old values of the updated tuple, respectively. For example, the *DistributeWealth* rule above would be written in POSTGRES as:

define rule DistributeWealth
on delete to employee
then update employee (employee.salary = employee.salary + .1 ∗ **old**.salary)

In Starburst, a single rule triggering may involve arbitrary combinations of inserted, deleted, and updated tuples (as will be described in Section 4). 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:[2]

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

3.5 Rule Ordering

When we discuss the semantics of run-time rule processing in Section 4, it will be seen that one important aspect, also present in expert systems rule languages, is *conflict resolution*: the choice of which rule to execute when multiple rules are triggered. In many active database systems this choice is made more or less arbitrarily; however, some database production rule languages provide features whereby the rule definer can influence conflict resolution.

Ode and RPL provide no language features for 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. (Further details on Ariel's conflict resolution strategy are given in Section 4.1.) HiPAC departs from other active database systems in that multiple triggered rules are executed concurrently (see Section

[2]In our examples we try to approximate the query language of the system being described. For example, Starburst uses SQL while Ariel uses Quel. Regardless, all of the examples should be understandable even for readers unfamiliar with database query languages.

4.2). Even so, HiPAC includes a mechanism whereby rules can be relatively ordered to influence the serialization order of concurrent execution.

4 Rule Processing 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; it also determines 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 essential. As with the rule language itself, there are a number of alternatives for rule execution, and different database rule systems have taken different approaches. We begin by describing the *recognize-act cycle*, which is the semantics used by most AI production rule systems, including OPS5. We then consider extensions and alternatives to this semantics taken by various database rule systems. We also briefly discuss how active database systems recover from errors during rule execution.

In expert systems, rules usually are processed using the following algorithm, known as the recognize-act cycle:

> *initial match* (test rule conditions)
> **repeat until** no rules match or *halt* is executed
>   perform *conflict resolution* (pick a triggered rule)
>   *act* (execute the rule's action)
>   *match* (test rule conditions)
> **end**

In the *match* phase, rule patterns are matched against data in the working memory to determine which rules are triggered and for which instantiations. The entire set of triggered rule instantiations is called the *conflict set*, and one instantiation is chosen from this set using a *conflict resolution strategy*. In the *act* phase, the selected rule's action is executed for the selected instantiation, then the cycle repeats.

As explained in Section 3, RPL has modified the OPS5 language for the database setting by replacing pattern-oriented rule conditions by database queries. Each tuple produced by the query in a rule condition is an instantiation for that rule. Rule processing in RPL uses the following variation on the recognize-act cycle:

initial match (execute rule conditions)
repeat until no rule conditions produce tuples
    perform *conflict resolution* (pick a triggered rule)
    act (execute the rule's action for each tuple produced by the condition)
    match (execute rule conditions)
end

Notice that here, in a single *act* phase, the selected rule's action is executed for all instantiations of the rule, rather than for only one instantiation as in the original recognize-act cycle given above. Firing the selected rule for all instantiations in each phase rather than only one is sometimes known as firing rules *to saturation* [12]. The semantic and practical differences between firing single instantiations and firing rules to saturation are explored in RDL1, a production rule implementation of a deductive database system [12, 30]. A related modification to the recognize-act cycle is *set-oriented* firing, in which the selected rule's action is executed once for the entire set of instantiations [21]. Ariel and Starburst both use a form of set-oriented firing; see Sections 4.1 and 4.5.

In active database systems, where production rule processing is fully integrated with conventional database activity (e.g. queries, modifications, transactions), a pure recognize-act semantics is not always appropriate or adequate. Furthermore, in addition to the rule processing algorithm itself, it must be determined what exactly during database activity causes rule processing to be invoked. For example, in relational systems, operations generally are performed in sets, e.g. a set of tuples is inserted into a table, or the set of tuples satisfying some condition is updated. Multiple operations generally are grouped into transactions. Hence, rule processing could be invoked by single tuple-level changes, by sets of changes corresponding to one or more operations, or by entire transactions. We call this the *granularity* of rule processing; this issue arises in object-oriented systems as well.

There is considerable variance in the semantics of rule processing taken by existing active database systems. In the remainder of this section we separately consider each of the projects we have been discussing and describe its approach to run-time rule processing. For convenience, in these descriptions we use the term database "user" to mean user or application program.

4.1 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 rule processing granularity 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. Of all the active database systems (excluding RPL), Ariel's rule processing is closest to that of expert systems rule languages; it uses a recognize-act cycle identical to that of OPS5.

Recall from Section 3.4 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 again the *MonitorNewBobs* 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 (matching) 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.

Regarding conflict resolution, recall from Section 3.5 that each Ariel rule is assigned a numeric priority, but that the assignments need not be unique. In the case of rules with the same priority, Ariel uses a mechanism similar to that used by OPS5 [4]. 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

One intent of Ariel's conflict resolution strategy is to simplify rule programming by causing rules to execute "at the right time" even after new rules have been added. In addition, selection criterion 2 helps enable rules to simulate loops and procedure calls.

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 then deleted, the net effect is deletion of the original tuple; if a tuple is inserted then updated, the net effect is insertion of the updated tuple; if a tuple is inserted then deleted, the net effect is no modification at all. Further details on Ariel's rule language and rule processing semantics appear in [24].

4.2 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. [19, 37]. Coupling modes determine how rule events, conditions, and actions relate to database transactions. Whereas in Ariel (and most 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.

Let E, C, and A denote the events, conditions, and actions of rules, respectively. 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, or *decoupled*, indicating execution in a separate transaction. Not all combinations of coupling modes make sense; Figure 1 shows the seven combinations that are

	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

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.

Rule processing in HiPAC is invoked whenever any event occurs that triggers one or more rules. As mentioned in Section 3.5, 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 [35], 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 3.5, HiPAC rules may have relative ordering, and this ordering is used to influence the serialization order of concurrently executing nested sub-transactions. Further details on HiPAC's rule language and rule processing semantics appear in [9, 11, 32].

4.3 Ode

As described in Section 3.2, there are two types of rules in Ode, constraint rules and trigger rules, hereafter referred to as *constraints* and *triggers*. Constraints and triggers have very different execution semantics, so we describe each in turn.

Recall that in Ode, constraints are associated with a class, are triggered by any method invocation on any object in that class, and consist of a condition and a single action (where the action is executed if the condition is false). As an example, the following constraint specifies that for each object in the *employee* class, if the salary is greater than 100 then it should be set to 100:

class employee
...
  constraint: salary $\leq$ 100 $\rightarrow$ salary = 100

Each constraint is specified as either *hard* or *soft*. All hard constraints triggered by a method invocation are processed immediately following the method invocation (in HiPAC terminology, they have immediate E-C coupling mode). All soft constraints triggered by a method invocation are processed at the end of the current transaction (in HiPAC terminology, they have deferred E-C coupling mode). The basic algorithm for processing a set of constraints in Ode is described as follows:

For each triggered constraint C_1, C_2, ..., C_n:
1. Evaluate C_i's condition
2. If the condition is false, execute C_i's action
3. Evaluate C_i's condition again
4. If the condition is still false, abort the transaction

The ordering of C_1, C_2, ..., C_n is arbitrary. Note that action execution (step b) can invoke a method that recursively calls this rule processing algorithm (for the same or for different objects and constraints). Also note that this rule processing semantics is particularly designed to be used for enforcing database integrity constraints: either the rules successfully establish a state in which all rule conditions are true, or the transaction is aborted.

Recall that like constraints, Ode triggers are associated with a class and consist of a condition and a single action. However, a trigger cannot actually be triggered until it is activated on a particular object; once activated on an object, a trigger is triggered by any method invocation on that object. As an example, the following trigger can be activated on objects in the *department* class; it reduces the department's budget if the number of employees falls below a threshold specified by the trigger's activation parameter:

class department
...
  trigger: budget(threshold): num-employees $<$ threshold $\rightarrow$ reduce-budget(**this**)

(When the trigger's action executes, "**this**" refers to the object for which the rule is triggered.) When a method invocation triggers one or more triggers, each of the trigger's conditions is evaluated. For those triggers whose conditions are true, when the current transaction commits a separate transaction is spawned to execute the trigger's action. (In HiPAC terminology, triggers have immediate E-C coupling mode and a variation of decoupled C-A coupling mode.) Ordering of trigger condition evaluation is irrelevant since no additional operations are performed during trigger processing. Once activated, a trigger can be deactivated; if a trigger is designated as *once-only* then it is deactivated automatically after it is triggered. Further details on Ode's rule language and rule processing semantics appear in [19].

4.4 POSTGRES

In POSTGRES, unlike in other relational active database systems, 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 set-oriented as in Ariel and Starburst.) 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 the recognize-act cycle used by the other relational systems. 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. Further details on POSTGRES's rule language and rule processing semantics appear in [40, 41].

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 (and Starburst), this rule is triggered only once, so the salaries of the new employees are the same.

4.5 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.[3]

[3]It is interesting to note, however, that Ariel's default is the minimum granularity while Starburst's default is the maximum.

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, and 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 4.1).

Starburst rule processing uses the following variation on the recognize-act cycle:

> *initial match* (find triggered rules based on initial set of changes)
> **repeat until** no rules are triggered
>   perform *conflict resolution* (pick a triggered rule)
>   evaluate the rule's condition
>   *act* (if the condition is true, execute the rule's action)
>   *match* (find additional triggered rules based on new changes)
> **end**

One important difference here is that in the *match* phase, Starburst determines all 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. Regarding 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. If the rules are totally ordered then conflict resolution is completely deterministic; if the rules are not ordered at all, then conflict resolution is completely arbitrary.

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 usual, if **rollback** is executed in a rule action, then rule processing terminates and the transaction is aborted. Further details on Starburst's rule language and rule processing semantics appear in [45, 46].

4.6 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—e.g., because data it depends on (such as a table) has been deleted, because data access privileges have been revoked, because concurrently executing transactions have created a *deadlock* [43], 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 takes some of these possibilities into account [9]. Each of these alternatives seems reasonable in various contexts; at the minimum, transaction abort in response to errors is a reasonable expectation for any database rule system.

5 Implementation Issues

Active database systems must support all of the features provided by conventional database systems, including *data definition* (to describe the format of the data), *data manipulation* (to perform queries and modifications), storage management, transaction management, concurrency control, and crash recovery (recall Section 2). In addition, active database systems must provide mechanisms for rule triggering and condition testing during database and/or rule processing, for rule action execution,

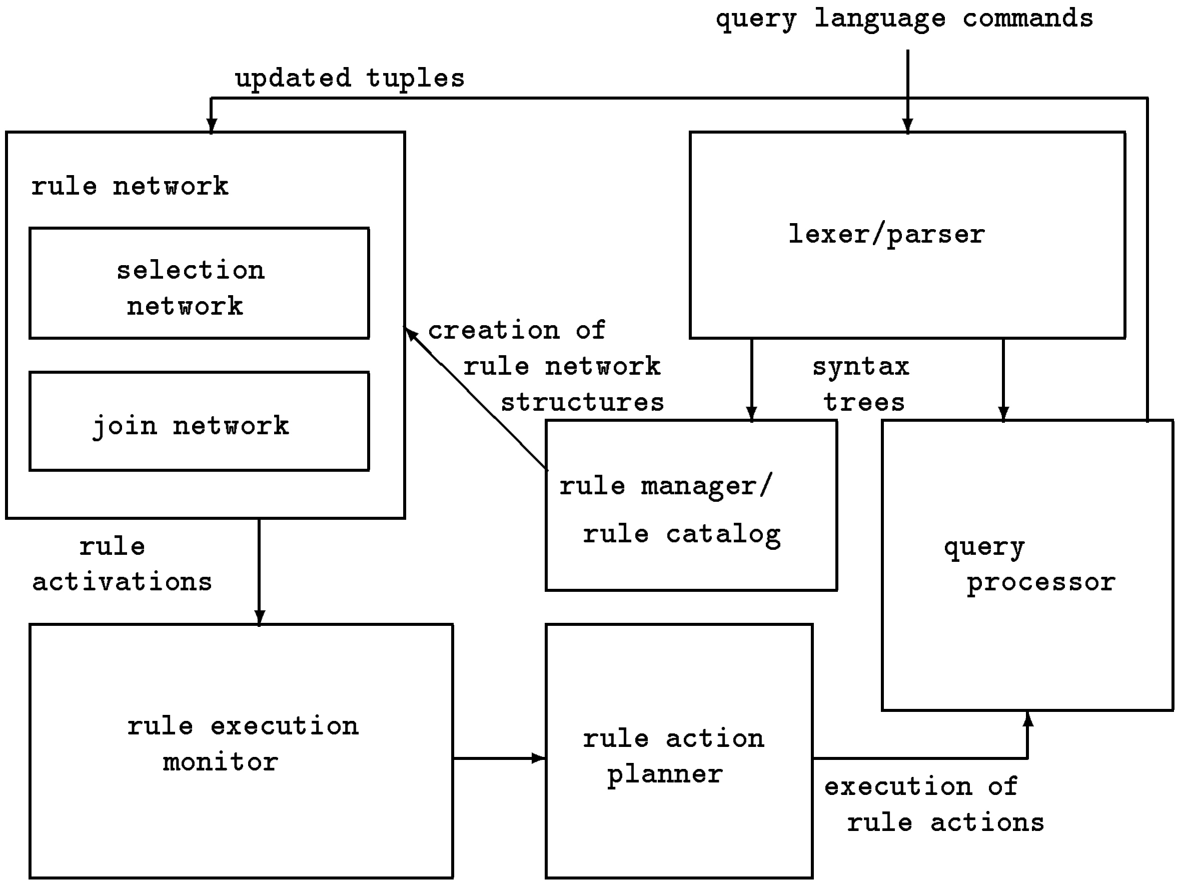

Figure 2: Architecture of the Ariel system

and for user development of rule applications. In this section we describe the issues that arise and the approaches that have been taken when integrating these features into a working database management system (DBMS).

5.1 Ariel Architecture

As an example, we begin by describing the architecture of the Ariel system, depicted in Figure 2. The *lexer/parser* and *query processor* shown in the diagram function as they do in a conventional DBMS, processing data definition and data manipulation commands. When data modification commands are executed, the modified tuples are packaged as *tokens* and passed to the *rule net-*

work, where rule conditions are tested. The *rule manager/rule catalog* handles rule definition and manipulation tasks. The *rule execution monitor* maintains the set of triggered rules and schedules their execution. Finally, the *rule action planner* 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. Further details of rule condition testing and action execution in Ariel are described in Sections 5.3 and 5.4.

Next, we discuss implementation characteristics of the other active database systems covered in this survey, particularly as they differ from Ariel. We then turn to more detailed discussions of rule condition testing, rule action execution, and programming support for rule developers.

5.2 Implementation Characteristics of Other Systems

In RPL, rule processing is implemented on top of a commercial relational DBMS, rather than integrated into a DBMS. (In the database context, this usually is referred to as *loosely coupled* rather than *tightly coupled* rule processing.) Rule conditions are tested by submitting queries to the underlying DBMS and—after the initial match phase—comparing the results with previous results from the same query. Rule actions are executed by submitting data manipulation commands to the underlying DBMS. While this approach is sufficient for experimenting with a database rule language, it does not provide the functionality or efficiency of the fully integrated approach. Further details on the implementation of RPL appear in [14].

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 with 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 [41]. Further details on the implementation of rules in POSTGRES appear in [40, 41].

The Starburst DBMS has as one of its primary goals *extensibility* [23], 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 3.4) 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. Further details on the implementation of rules in Starburst appear in [45].

While there are three different implementations of HiPAC—one in C, one in Smalltalk-80, and one in Lisp—all are main-memory prototypes not fully integrated with a conventional DBMS. The most substantial of these is the Smalltalk 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 DBMS operations, and rules running inside the DBMS can invoke application operations. Using this implementation, a simulated financial trading application was coded, with most control of the application embedded in rules. Further details on HiPAC's implementation and applications appear in [9, 32].

Rule processing in Ode is built into the implementation of O++, a database programming language extension to C++. For each Ode class, the hard constraints, soft constraints, and triggers defined for that class are packaged into three system-defined methods: `hard_constraints()`, `soft_constraints()` and `triggers()`. Whenever a user-defined method is invoked on an object in the class, it is followed by invocation of each of the three system-defined methods. The `hard_constraints()` method checks constraint conditions and executes actions as described in Section 4.3. The `soft_constraints()` method places the object on a "to be checked" list; at the end of the transaction, soft constraints are checked for each object on the list. The `triggers()` method checks trigger conditions and, if they are satisfied, places trigger actions on a "to be executed" list; when the transaction reaches its commit point (following the checking of any soft constraints), the actions on this list are scheduled for execution in separate transactions. Further details on the implementation of rules in Ode, including a number of suggested refinements, appear in [19].

5.3 Condition Testing

In many active database systems, including HiPAC, Ode, RPL, and Starburst, rule conditions are tested during rule processing by querying the database. Although this is a straightforward approach, there is clear potential for poor performance. In the tuple level implementation of rules in POSTGRES, markers are placed only on tuples satisfying rule conditions, so conditions need not be tested during rule processing. However, the overhead of condition testing still occurs, in the maintenance of markers through data modifications. The most sophisticated approach to condition testing in a database rule system is taken in Ariel, so we outline Ariel's condition testing mechanism here. Further details appear in [24, 25].

Ariel's condition testing mechanism uses an algorithm called *A-TREAT*, which is a descendent of the *TREAT* algorithm used in some main-memory production rule systems [33]. TREAT itself is a descendant of the pioneering *Rete* algorithm [18], developed for OPS5. These algorithms use *discrimination networks* to efficiently compare large collections of patterns to large collections of data without iterating over the collections. *Tokens* representing modified data items are passed through the networks. When a token emerges at the "bottom" of a network, the algorithm deduces that a rule is triggered for the data represented by that token. The TREAT approach modifies the original Rete network for improved performance, and the A-TREAT approach modifies the TREAT network for the database context. We briefly describe each of these in turn.

In Rete networks, there are six types of nodes:

- one *root* node
- *t-const* nodes, that test selection conditions (i.e. simple predicates)
- *α-memory* nodes, that store the results of t-const nodes

- *and* nodes, that join tokens from two α-memory and/or β-memory nodes
- *β-memory* nodes, that store the results of *and* nodes
- *p-nodes*, that hold tokens matching an entire rule condition

Tokens enter the network at the root node; they pass through, are stored at, or are eliminated by nodes in the network according to the type of the node and the presence of other tokens in the network. When a token reaches a p-node, it enters the corresponding rule's conflict set. Further details on the Rete algorithm can be found in [18].

An advantage of the Rete algorithm is its ability to reuse temporary results; a disadvantage is its need to maintain and store the contents of β-memory nodes. The TREAT algorithm eliminates the use of β-memory nodes; for details see [33]. A simulation study has shown that TREAT can be expected to perform better than Rete in the context of database rule systems [44]. TREAT has also been shown to outperform Rete for a collection of OPS5 applications [33], although Rete can perform better in certain situations.

Ariel's A-TREAT algorithm is designed to both speed up rule processing in a database environment and reduce the storage requirements of TREAT. An important performance optimization in A-TREAT is a *top-level discrimination network* for testing single-table selection conditions. This top layer uses a special index optimized for efficiently testing large numbers of selection conditions. This index makes use of an *interval binary search tree* to efficiently test conditions that specify closed intervals (e.g. $const_1 < table.column \leq const_2$), open intervals (e.g. $const < table.column$), or points (e.g. $const = table.column$). Details of this mechanisms can be found in [25]. A different data structure called the *interval skip list* can be used in place of the interval binary search tree, with similar performance and a more straightforward implementation [26].

As a second optimization, Ariel can reduce the amount of state information stored in its discrimination network by replacing α-memory nodes with *virtual* α-memory nodes. Virtual α-memory nodes contain only the predicate associated with the node, not the tuples matching the predicate. This can be crucial in the database setting, since if most tuples satisfy the predicate associated with an α-memory node, then the node may need to store almost as much data as is stored in the database itself (which may be impossible). When a virtual α-memory node is accessed, the predicate stored in the node is processed to derive the value of the node; details are given in [24].

Finally, in addition to the performance enhancement techniques mentioned above, Ariel has generalized the notion of tokens and α-memory nodes from the standard TREAT network in order to effectively test both event- and transition-based conditions with a minimum of restrictions on how such conditions can be used [24].

5.4 Action Execution

In all database rule systems, rule actions are executed by submitting operations to the DBMS query processor. However, most systems need some additional mechanism for binding the data that triggered a rule to operations in the rule's action. Each active database system takes a somewhat different approach to this. Here we briefly survey the various approaches; details can be found in the references for each system's implementation given in Sections 5.1 and 5.2 above.

In Ariel, at the time a rule is scheduled for execution, the tuples whose modification triggered the rule are stored in the p-node for that rule (see Section 5.3). Recall that in Ariel a reference in the rule action to a table specified in the rule condition implicitly references the triggering data. Hence, when a command in the rule action is executed, tuples for the rule's trigger table are derived by scanning the p-node rather than accessing the database table itself. In POSTGRES, the binding problem is relatively simple since rules are triggered by changes to a single tuple. In the query rewrite implementation, the notion of binding is built into the query rewrite process itself; in the tuple level implementation, the triggering tuple is provided to the query processor along with the rule action so references to **new** and **old** can be evaluated. Recall that in RPL rule processing takes place outside the DBMS (Section 5.2). The rule processor determines the tuples matching a rule by performing database queries; identifiers for these tuples are then submitted as part of the command to execute the rule's action. In Starburst, the triggering data is accessed through transition tables, which are materialized during query processing as described in Section 5.2. In Ode, a rule is triggered for a particular object, and the rule's action is executed with that object as the "current" object, i.e. as the object referenced through keyword **this** (see Section 4.3 for an example). Finally, in HiPAC triggering data is passed to the rule's action through explicitly specified parameters.

5.5 Rule Programming Support

Programming tools for expert systems rule languages have evolved to include many useful features that support the rule programmer. Many of these features, such as those found in OPS5 [4] and KEE [27], would be valuable for database rule programming as well. These 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. Simple versions of some of these features exist in a few database rule systems; more sophisticated and complete versions will certainly emerge as active database systems mature over time.

A number of additional programming support features are valuable in database rule systems due to the on-line nature of database applications and the fact that data may be shared by many concurrent applications. These features include the ability to control errors in rule programs (such as the failure of rule processing to terminate in a timely manner), 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. (Unlike most expert systems applications, database applications often must be available continuously for an indefinite period of time, and it may not be possible to shut down the system to add new rules or fix bugs.) Again, simple versions of some of these features exist in a few database rule systems, but more sophisticated and complete versions are needed and will emerge.

In the remainder of this section we outline several important rule programming features in active database systems, explaining what currently exists and what can be expected in the future.

5.5.1 Rule Creation, Deletion, Activation, and Deactivation

All active database systems support creation and deletion of individual rules. In some systems, rule creation and deletion can occur while the database system is processing other user transactions (sometimes requiring a special concurrency control mechanism); in other systems, it is assumed that rules are created and deleted off-line. For logically grouping together rules associated with a particular application, POSTGRES and Starburst have introduced the notion of *rule sets*. Rule sets can be created and deleted, and rules can be added to and removed from sets. In POSTGRES, rule sets provide a mechanism whereby groups of rules can be *activated* and *deactivated* with one command. (Rule activation and deactivation are discussed below.) In Starburst, rule sets provide a mechanism whereby rule processing can be invoked only for the rules in a particular set; recall Section 4.5. Note that for object-oriented systems in which each rule is treated as a first-class object,[4] the usual structuring mechanisms of object-orientation (e.g. classes and hierarchies) are available to rules as well as to data.

A useful mechanism in an active database system is the ability to explicitly *activate* a rule (i.e. make the system start monitoring the rule's event and condition, and make the rule eligible to be executed) and *deactivate* a rule (i.e. make the system stop monitoring the rule's event and condition, and make the rule ineligible to be executed). Because rules are a persistent part of the database and have a potentially long lifespan, such a mechanism can greatly facilitate the task of the rule programmer and the database administrator. Activation and deactivation are provided in several systems; in fact, in some systems (such as Ariel and Ode), a created rule is not eligible to be triggered until it is activated. Certain semantic issues must be addressed with respect to activation and deactivation, particularly for pattern-based rules. For example, when a pattern-based rule is activated, it could be run immediately if its condition matches existing data, or it could be run only when new data satisfies its condition. The choice between these can affect the behavior of a rule application in subtle but important ways.

5.5.2 Querying the Set of Rules

Managing a collection of database rules can be a challenging task. When the collection becomes relatively large, even simply locating a desired rule can be difficult. To effectively manage a large collection of rules, mechanisms are needed for posing queries against the rules. Most active database prototypes treat rules as named system objects (similar to tables) and provide simple commands for retrieving individual rules or rule sets by name. As with rule structuring, object-oriented systems have an advantage here if rules are treated as first-class objects (as in HiPAC): rules can be queried using the standard query language for objects, rather than through a separate query language provided for rules.[5]

Even when rules are stored as first-class objects, and certainly when they are stored as system objects, the functionality provided by the query language may not be enough to express all of

[4] Of the systems covered in this survey, only HiPAC treats rules in this way.

[5] Unfortunately, treating rules as first-class objects has a tendency to result in an awkward notation for rule creation, but this is a syntactic problem that can be overcome by providing an additional layer for rule creation.

the interesting queries on rules. For example, often it may be desirable to express queries that cross-reference rules and data, such as:

> Which rules refer to column *salary* of table *employee* in their condition?
>
> Which rules modify column *budget* of table *department* in their action?

This kind of query cannot be carried out without examining the internal structure of rule conditions and actions. Some expert systems tools have utilities that support such cross-referencing, and similar features would be useful in the database context.

5.5.3 Limiting Forward-Chaining

Rule processing is subject to infinite loops (i.e. rules may trigger each other indefinitely), and 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. 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 [1]. A run-time solution to detecting and preventing infinite loops is to provide a forward-chaining (i.e. rule triggering) depth 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. Unfortunately, it always may be possible for correct rule executions to exceed the limit, for example if rules are being used to traverse arbitrarily large list or graph data structures.

6 Conclusions, Applications, and Future Directions

This survey has described the state-of-the art in active database systems, including rule language design, rule processing semantics, implementation issues, and programming support. Active database systems represent a unique merging of traditional passive database systems and AI production rule processing technology. Production rules in 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:

Support for application development: In Section 5.5 we described a number of features, not present in most 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 [5, 7, 8]. 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; this is discussed below.

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 languages while maintaining an efficient implementation certainly deserve further study.

Smooth integration with the DBMS: Database rule systems interact strongly with the query processing, authorization, concurrency control, and crash recovery mechanisms of the DBMS. For rule processing to function successfully in a large on-line DBMS, the rule system must be carefully and fully integrated with each of these conventional database system components.

Improved algorithms: Highly efficient algorithms for processing rules, particularly for rule condition testing, are crucial for delivering the functionality of active databases without excessively degrading the performance of conventional query processing. While some work has been done in this area, continued improvements are needed.

Applying parallelism: For some database rule languages, parallelism in rule condition testing may be necessary to achieve desired levels of performance. Parallelism has been used successfully to increase the performance of the OPS5 rule language both on a shared memory multiprocessor [22] and on a fine-grain parallel machine [29]. This work can serve as a starting point for parallel rule condition testing in active databases.

Feedback from the initial use of rule processing in large-scale database applications should provide valuable guidance to help researchers and practitioners continue to improve the capabilities of active database systems.

As active databases begin to be used more widely, we are likely to see the development of new kinds of database applications in which the application program and the database rule processor participate equally in the computation (rather than the database system acting as a "slave" to the application, which is the conventional mode of interaction). Initial experimentation with such applications has been performed in the context of HiPAC [32], and the commercial Ingres system includes rule processing facilities and a mechanism for "asynchronous multicast" that make such applications possible in practice [2]. Examples of potential application domains using this approach include:

Real-time information display: For example, an application might create a new window on a stock trader's workstation whenever the price of a certain stock exceeds a threshold. Other examples include timely displays of aircraft flight information for an air traffic controller, or display of information related to the flow of gas in a pipeline.

Intelligent situation monitoring: This might be used, for example, in a law enforcement database, to correlate different crime reports as they are added to the database.

Immediate applications: Traditional "batch" applications that run periodically (e.g. every day, week, or month) could be redesigned to run automatically and immediately whenever needed. For example, rules could be used in a warehouse database to automatically reorder items when their stock level drops too low.

In conclusion, production rules in database systems have the potential to improve existing database applications and to support new ones that are not now feasible. Moreover, database production rules can extend the reach of knowledge-base and expert systems to let them monitor important conditions over large, permanent on-line databases. The incorporation of production rules into database systems is a promising new technology, and this technology is quickly emerging in research prototypes, commercial systems, and applications.

References

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

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

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

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

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

[6] S. Ceri, G. Gottlob, and L. Tanca. *Logic Programming and Databases.* Springer-Verlag, Berlin, 1990.

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

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

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

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

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

[12] C. de Maindreville and E. Simon. A production rule based approach to deductive databases. In *Proceedings of the Fourth International Conference on Data Engineering*, February 1988.

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

[14] L. M. L. Delcambre and J. N. Etheredge. A self-controlling interpreter for the relational production language. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, June 1988.

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

[16] Digital Equipment Corporation. *Rdb/VMS – SQL Reference Manual*, November 1991.

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

[18] C. L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19:17–37, 1982.

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

[20] N. Gehani, H. V. Jagadish, and O. Shmueli. Event specification in an active object-oriented database. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, June 1992.

[21] D. N. Gordin and A. J. Pasik. Set-oriented constructs: From Rete rule bases to database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, May 1991.

[22] A. Gupta. *Parallelism in Production Systems.* Pitman Publishing, 1987.

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

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

[25] E. N. Hanson, M. Chaabouni, C.-H. Kim, and Y.-W. Wang. A predicate matching algorithm for database rule systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, May 1990.

[26] E. N. Hanson and T. Johnson. The interval skip list: A data structure for finding all intervals that overlap a point. Technical Report TR92-016, Computer and Information Sciences Department, University of Flordia, Gainesville, Florida, June 1992.

[27] S. Hedberg and M. Steizner. *Knowledge Engineering Environment (KEE) System: Summary of Release 3.1.* Intellicorp Inc., July 1987.

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

[29] M. A. Kelly and R. E. Seviora. An evaluation of DRete on CUPID for OPS5. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, 1989.

[30] G. Kiernan, C. de Maindreville, and E. Simon. Making deductive database a practical technology: A step forward. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, May 1990.

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

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

[33] D. P. Miranker. TREAT: A better match algorithm for AI production systems. In *Proceedings of the AAAI Conference on Artificial Intelligence*, August 1987.

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

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

[36] ORACLE Corporaton. *ORACLE7 Reference Manual*, 1992.

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

[38] T. Sellis, C.-C. Lin, and L. Raschid. Implementing large production systems in a DBMS environment: Concepts and algorithms. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, June 1988.

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

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

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

[42] A. Tzvieli. On the coupling of a production system shell and a DBMS. In *Proceedings of the Third International Conference on Data and Knowledge Bases*, June 1988.

[43] J. D. Ullman. *Principles of Database and Knowledge-Base Systems, Volumes I and II.* Computer Science Press, Rockville, Maryland, 1989.

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

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

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