Active Database Systems

Norman W. Paton and Oscar Díaz†

Department of Computer Science, University of Manchester, Oxford Road, Manchester M13 9PL, UK. e-mail: norm@cs.man.ac.uk
†Departamento de Lenguajes y Sistemas Informaticos, University of the Basque Country, San Sebastián, Spain. e-mail: jipdigao@si.ehu.es

Active database systems support mechanisms that enable them to respond automatically to events that are taking place either within or outside the database system itself. Considerable effort has been directed towards improving understanding of such systems in recent years, and many different proposals have been made and applications suggested. This high level of activity has not yielded a single, agreed standard approach to the integration of active functionality with conventional database systems, but has led to improved understanding of active behavior description languages, execution models and architectures. This paper presents the fundamental characteristics of active database systems, describes a collection of representative systems within a common framework, considers the consequences for implementations of certain design decisions, and discusses tools for developing active applications.

Categories and Subject Descriptors: M.2.3 [**Database Management**]: Languages—*Active Database Systems*

General Terms: Database Systems

Additional Key Words and Phrases: Active databases, events, relational databases, object-oriented databases.

1. INTRODUCTION

Traditionally, database systems have been viewed as repositories that store the information required by an application, and that are accessed either by user programs or through interactive interfaces. In such a context, a range of different tools and systems are used together to support the requirements of the application. However, database systems are beginning to be applied to a range of domains associated with highly complex information processing, ever more substantial quantities of data, or highly stringent performance requirements, in which the conventional multi-component environment has proved to be unsatisfactory. This has resulted in a trend in database research towards more of the functionality required by an application being supported within the database system itself, giving rise to database systems with more comprehensive facilities for modeling both the structural and the behavioral aspects of an application. Among the fields that have received attention in recent years with a view to enhancing the behavioral facilities of database systems are database programming, temporal databases, spatial databases, multimedia databases, deductive databases and active databases. This paper focuses upon the latter.

Traditional database management systems (DBMSs) are *passive* in the sense that commands are executed by the database (e.g. query, update, delete) as and when requested by the user or application program. However, some situations cannot be effectively modeled by this pattern. As an example, consider a railway database where data is stored about trains, timetables, seats, fares and so on, which is accessed by different terminals. In some circumstances (e.g. public holidays, cultural events) it may be beneficial to add additional coaches to specific trains if the number of spare seats a month in advance is below a threshold value. Two options are available to the administrator of a passive database system who is seeking to support this requirement. One is to add the additional monitoring functionality to all booking programs so that the above situation is checked for each time a seat is sold. However, this approach leads to the semantics of the monitoring task being distributed, replicated and hidden among different application programs. The second approach relies on a polling mechanism that periodically checks the number of seats available. Unlike the first approach, here the semantics of the application is represented in a single place, but the difficulty stems from ascertaining the most appropriate polling frequency. If too high, there is a cost penalty. If too low, the reaction may be too late (e.g. the coach is added, but only after several customers have been turned away).

Active databases support the above application by moving the reactive behavior from the application (or polling mechanism) into the DBMS. Active databases are thus able to monitor and react to specific circumstances of relevance to an application. The reactive semantics is both centralized and handled in a timely manner. An active database system must provide a knowledge model (i.e. a description mechanism) and an execution model (i.e. a runtime strategy) for supporting this reactive behavior.

A common approach for the knowledge model uses rules that have up to three components: an event, a condition and an action. The *event* part of a rule describes a happening to which the rule may be able to respond. The *condition* part of the rule examines the context in which the event has taken place. The *action* describes the task to be carried out by the rule if the relevant event has taken place and the condition has evaluated to true.

Most active database systems support rules with all three of the components described above; such a rule is known as an event-condition-action or *ECA-rule*. In some proposals the event or the condition may be either missing or implicit. If no event is given, then the resulting rule is a condition-action rule, or *production rule*. If no condition is given, then the resulting rule is an event-action rule.

At first glance, the introduction of active rules to a database system may seem like a straightforward task, but in practice proposals have been made that support widely different functionalities. Among the issues that distinguish proposals are the expressiveness of the event language, the scope of access to database states from the condition and action, and the timing of condition and action evaluation relative to the event. The functionality of a specific system will be influenced by a number of factors, including the nature of the passive data model that is being extended, and the categories of application to be supported.

1.1 Active Database Applications

As mentioned above, database research often aims to extend the range of facilities within the database system for representing application concepts. Hence, additional capabilities are largely dependent on the applications that are being targeted. In the case of active rules, the following categories of application can be distinguished:

Database System Extensions. Active rules can be used as a primitive mechanism for supporting the implementation of other parts of a database system. For example, ECA rules have been used to support integrity constraints [Ceri et al. 1990; Diaz 1992], materialized views [Stonebraker et al. 1990; Widom et al. 1991], derived data [Etzion 1993], coordination of distributed computation [Dayal et al. 1990; Ceri and Widom 1993], transaction models [Geppert and Dittrich 1994], advanced data modeling constructs [Paton et al. 1993] and automatic screen updating in the context of database change [Diaz et al. 1994; Paton et al. 1996].

Such extensions to core database functionality are usually supported by defining a high-level syntax for the extended functionality, plus a mapping onto sets of active rules. For example, [Ceri et al. 1990] presents a constraint language for implementation using active rules, illustrated using an application that models a power distribution system. In this constraint language, the restriction that *no wire has a voltage that is greater than that of its type* is expressed thus:

```
wire:voltage > any (select max-voltage
                    from wire-type
                    where type = wire.type)
```

This constraint can be violated by a range of different update operations (e.g. a new `wire` is created of an existing `wire-type`, the `max-voltage` of a `wire-type` is updated, etc), that can then be monitored by a set of system generated active rules. For example, to check for violation of the constraint on insertion of a new `wire`, the following active rule could be used:

```
on insert into wire
if insert.voltage > any (select max-voltage
                         from wire-type
                         where type = insert.type)
do <action>
```

Here, the `on` clause defines the event (the `insert` of a tuple into the `wire` relation), the `if` clause expresses the condition, and the `do` clause specifies the action. Information about the event is referred to in the condition by accessing the `voltage` and `type` of the newly inserted tuple using the reserved word `insert`. In this example, the action could be defined in different ways – the update operation could be blocked by aborting the transaction, the constraint could be repaired by changing the voltage of the inserted wire, etc.

Closed Database Applications. This category of application involves the use of active functionality to describe some of the behavior to be manifested by a software system without reference to external devices or systems. For example, rules might be used to describe repair actions in a modeling database, to monitor sales in a stock control database, to propagate load calculations in an architectural design database, or to anticipate market activity in a portfolio management database. In these applications there may not be any mapping from a higher-level description onto the active rule language – ECA rules are used directly to support the semantics of the application. For example, in a portfolio management database a rule could be written that deletes any stock holders whose portfolios have value 0, while at the same time recording these holders in a distinct relation:

```
on update to value of Holder
if new.value = 0
do begin
    delete from Holder where reg# = update.reg# ;
    insert into VoidHolder
    values (update.reg#, update.name, update.address, today)
end
```

In this example, the event monitors updates to the `value` attribute of the `Holder` relation. The condition then checks the `new` element that has been assigned to the `value` attribute to see if it is `0`. If so, then the action is executed, which deletes the updated tuple from the `Holder` relation, and then inserts information from the deleted tuple into the `VoidHolder` relation. It is worth noting that both the condition and the action of this rule require access to information on the update that triggered the rule.

Open Database Applications. In this category of application, a database is used in conjunction with monitoring devices to record and respond to situations outside the database. For example, rules could be used in command and control applications to respond to evolving battlefield scenarios [Dayal et al. 1988], in medical applications to warn physicians of changes in a patient's condition [Blue et al. 1988], in transport applications to anticipate traffic holdups, and in air-traffic control to detect potentially dangerous aircraft movements [Naqvi and Ibraham 1994]. For example, in an aircraft monitoring database, the following rule adapted from [Naqvi and Ibraham 1994] could inform a controller when two aircraft are approaching each other:

```
on update to pos of aircraft
if exists (select *
           from aircraft Other
           where distance(Other.pos,new.pos) < 5000 and
                 distance(Other.pos,old.pos) > 5000)
do <send message to controller>
```

In this example, the event being monitored is the position of an aircraft notified to the database from an external device, and the action taken is a change to a display that the air traffic controller is monitoring. Both the `new` value and the `old` value for the `pos` affected by the event are accessed from within the condition.

1.2 Outline of Paper

This paper provides an overview of recent research into active database systems. Section 2 introduces an example application that is used throughout the paper. Section 3 presents the structural characteristics of active rules, and section 4 shows how different execution models can be used to characterize the run-time interpretation of a set of rules. Section 5 indicates what facilities may be available for managing rule bases, and section 6 describes and compares a range of representative active

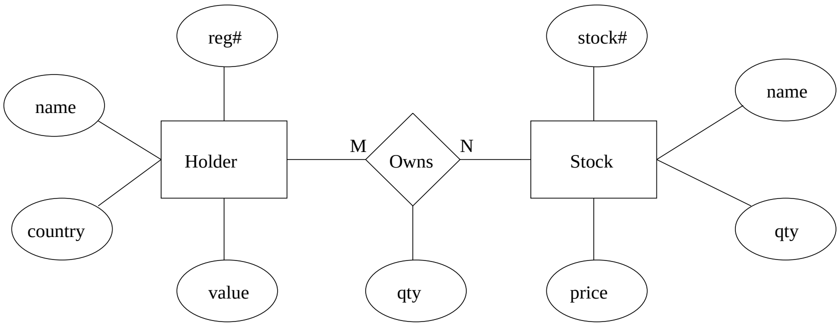

Fig. 1. Entity/Relationship diagram for portfolio database.

database systems within the framework presented in sections 3 to 5. Section 7 indicates what architectural features are important for the implementation of an active database system and section 8 considers the facilities that are useful for supporting the development of applications using active functionality. Section 9 presents some conclusions.

2. EXAMPLE APPLICATION

This section introduces a portfolio management database that is used throughout the paper to exemplify the functionality of active database systems [Chandra and Segev 1994]. In fact, a range of different financial applications stand to benefit from the presence of active functionality for monitoring financial transactions, identifying unusual patterns of activity, enforcing integrity constraints, maintaining derived data, generating timely reports, and performing periodic processing. The relevant entities and relationships are depicted graphically in figure 1.

In this example, a `Holder` is an individual or organization that owns stocks. Every `Holder` has a unique registration number, a `name`, a `country` and a total `value` of stock held. An organization that has been floated on the stock market is represented by the entity type `Stock`, and has attributes that record its `name`, share `price`, the total number of shares available, and the unique identification number by which it can be referenced. The `Owns` relationship indicates that a `Holder` possesses `qty` items of a particular kind of `Stock`. A relational schema for implementing this database using SQL is presented in figure 2.

Specific examples of active behavior that can be used in this application are introduced when they are used to illustrate concepts, rather than presented as a group here. Where rules are presented in this paper, the syntax used is not that of any specific active rule system, but rather a notation based upon SQL that should require minimal explanation.

3. KNOWLEDGE MODEL

The knowledge model of an active database system indicates *what* can be said about active rules in that system. This is in contrast with the execution model, which determines *how* a set of rules behaves at runtime, as presented in section 4. As the knowledge model essentially supports the description of active functionality, the features dealt with in this section will often have a direct representation within

```
create table Holder (
        reg#    int,
        name    char(30),
        country char(30),
        value   real
)

create table Stock (
        stock#  int,
        name    char(30),
        price   real,
        qty     int
)

create table Owns (
        reg#    int references Holder(reg#),
        stock#  int references Stock(stock#),
        qty     int
)
```

Fig. 2. Relational tables for storing portfolio information.

the syntax of the rule language. Rather than using any particular rule language to illustrate features of the knowledge model, this section is based around a number of dimensions of active behavior, which extend those presented in [Paton et al. 1994]. These dimensions essentially make explicit the decision space within which the designers of active rule systems work, without endeavoring to provide any formal description of the semantics of specific rule systems, a topic that is dealt with in section 8.2.

The concepts considered in this section as dimensions are clearly not new – the aim is to provide a framework for characterizing active database functionality, rather than to introduce new notions, so the terminology used should be familiar to the readers of papers such as [Dayal et al. 1988; Widom and Finkelstein 1990; Stonebraker et al. 1990]. The dimensions of rule functionality considered in this paper are presented in a tabular form. In the tables, the symbol $\subset$ is used to indicate that the particular dimension can take on more than one of the values given, whereas $\in$ indicates a list of alternatives.

The knowledge model of an active rule is considered to have (up to) three principal components, an *event*, a *condition* and an *action*. The dimensions associated with these structural components of an active rule are presented in Table 1 and discussed in the following subsections.

3.1 Event

An event is something that happens at a point in time. Specifying an event therefore involves providing a description of the happening that is to be monitored. The nature of the description and the way in which the event can be detected largely depend on the **Source** or *generator* of the event. Possible alternatives are:

—**structure operation**, in which case the event is raised by an operation on some piece of structure (e.g. *insert* a tuple, *update* an attribute, *access* a tuple).

Table 1. Dimensions for the knowledge model.

Event	**Source** $\subset$ {Structure Operation, Behavior Invocation, Transaction, Abstract, Exception, Clock, External} **Granularity** $\subset$ {Member, Subset, Set} **Type** $\subset$ {Primitive, Composite } **Operators** $\subset$ {or, and, seq, closure, times, not } **Consumption mode** $\subset$ {Recent, Chronicle, Cumulative, Continuous } **Role** $\in$ {Mandatory, Optional, None}
Condition	**Role** $\in$ {Mandatory, Optional, None} **Context** $\subset$ {DB_T, $Bind_E$, DB_E, DB_C}
Action	**Options** $\subset$ {Structure Operation, Behavior Invocation, Update-Rules, Abort Inform, External, Do Instead } **Context** $\subset$ {DB_T, $Bind_E$, $Bind_C$, DB_E, DB_C, DB_A}

—**behavior invocation**, in which case the event is raised by the execution of some user-defined operation (e.g. the message *display* is sent to an object of type *widget*). It is common for event languages to allow events to be raised *before* or *after* an operation has been executed.

—**transaction**, in which case the event is raised by transaction commands (e.g. abort, commit, begin-transaction)

—**abstract** or **user-defined**, in which case a programming mechanism is used that allows an application program to signal the occurrence of an event explicitly (e.g. in response to some information entered by a user).

—**exception**, in which case the event is raised as a result of some exception being produced (e.g. an attempt is made to access some data without appropriate authorization).

—**clock**, in which case the event is raised at some point in time [Dayal et al. 1988; Gatziu and Dittrich 1994]. Absolute (e.g. the 13th of November 1998 at 15:00), relative (e.g. 10 days after the shares are sold) and periodic (e.g. the first day of every month) time events are reported in the literature.

—**external**, in which case the event is raised by a happening outside the database (e.g. the temperature reading goes above 30 degrees [Dayal et al. 1988]).

The **Event Granularity** of an event indicates whether an event is defined for every object in a **set** (e.g. every instance of a class), for given **subsets** (e.g. all staff members except professors) or for specific **members** of the set (e.g. to prevent unauthorized access to specific instances, or to enable the update of the specific widget objects that are presently on screen [Diaz et al. 1994]).

The **Type** of an event can be:

—**primitive**, in which case the event is raised by a single, low-level occurrence that belongs to one of the categories described in **Source** above. For example, the event `on insert to Owns` monitors the insertion of new tuples into the `Owns` relation.

—**composite**, in which case the event is raised by some combination of primitive or composite events using a range of operators that constitute the *event algebra.*

The range of event operators varies from system to system. The most common are: *disjunction* – $E_1 or E_2$ occurs when either E_1 or E_2 has occurred; *conjunction*

– $E_1 and E_2$ happens when both E_1 and E_2 have occurred in any order; *sequence* – seq(E_1, E_2) occurs when E_1 occurs before E_2; *closure* – *closure E in Int* is raised only once the first time E is signaled, regardless of later occurrences of E in the time interval Int; *history* – *times(n,E) in Int* is signaled when event E occurs n times during the time interval Int; $not - not E_1 in Int$ detects the non-occurrence of the event E_1 in the interval Int.

As an example of a rule with a composite event, the following rule enforces the constraint that the `qty` attribute of `stock` is the same as the amount recorded in the `Owns` relation:

```
on update to qty of Holder or update to qty of Stock or
   insert to Stock or delete to Stock or
   insert to Holder or delete to Holder
if exists (select *
           from Stock
           where qty <> (select sum(qty)
                         from Owns
                         where Owns.reg# = Stock.reg#)
          )
do abort
```

As a further example, to detect whether a stock price has changed during a working day the following event can be used: `on update to price of Stock in [09:00, 17:00]`.

Rich event algebras have been proposed for a range of systems, including HiPAC [Dayal et al. 1988], SAMOS [Gatziu and Dittrich 1994], ODE [Gehani et al. 1992] and Sentinel [Chakravarthy et al. 1994]. However, composite event handling presents challenges in terms of semantics and efficiency that have yet to be fully addressed.

When detecting composite events, there may be several event occurrences (of the same event type) that could be used to form a composite event. As an example, consider a composite event CE which is the sequence of events $EV1$ and $EV2$. If two occurrences of event $EV1$, first $ev1$ and later $ev1'$, have already been signaled, and an occurrence of event $EV2$, e.g. $ev2$, is now produced, there is a question as to what instances of CE should be raised. Possibilities include *sequence(ev1,ev2)* or *sequence(ev1',ev2)* or *sequence(ev1,ev2)* ∪ *sequence(ev1', ev2)*. The alternatives are distinguished using **consumption policies**. In [Chakravarthy et al. 1994] four possible consumption policies are introduced: a **recent** context, which considers the most recent set of events that can be used to construct the composition (in the previous example, *sequence(ev1', ev2)* is detected when *ev2* arises, after which *ev1'* and *ev2* are no longer considered for the detection of CE); a **chronicle** context, which consumes the events in chronological order (*sequence(ev1, ev2)* is signaled when *ev2* arises, after which *ev1* and *ev2* are no longer considered for the detection of CE); a **continuous** context, which defines a sliding window and starts a new composition with each primitive event that takes place (two sequence events would begin to be constructed when *ev1* and *ev1'* arise, and both sequence events would be signaled as *ev2* is detected); and a **cumulative** context, which accumulates all the primitive events until the composite event is finally raised (a sequence event is signaled only once when *ev2* arises, where the first parameter of the sequence

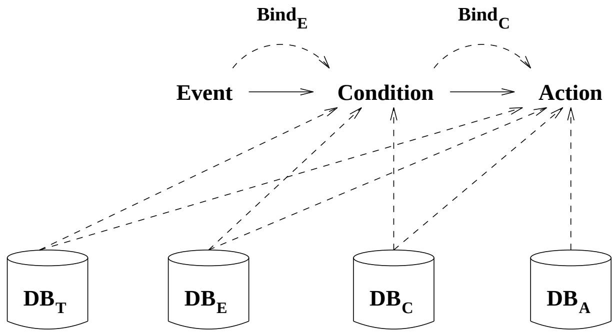

Fig. 3. The context within which a rule is processed.

includes the paramenters of all the occurrences of EV1, i.e. *ev1* and *ev1'*[1]). The rationale for each context can be found in [Chakravarthy et al. 1994].

The **Role** of an event indicates whether events must always be given for active rules, or whether the explicit naming of an event is not necessary. If the role is **optional**, then when no event is specified *condition-action* rules are supported, which have significantly different functionality and implementations from event-condition-action (ECA) rules, as described in section 7.5. If the role is **none** then events cannot be specified, and all rules are *condition-action* rules. If the role is **mandatory** then only ECA-rules are supported.

3.2 Condition

The **Role** of a condition indicates whether or not it must be given. In ECA-rules, the condition is generally **optional**. When no condition is given for an ECA-rule, or where the role is **none**, an *event-action* rule results. In systems in which both the event and the condition are optional, it is always the case that at least one is given.

The **Context** indicates the setting in which the condition is evaluated. The different components of a rule are not evaluated in isolation from the database or from each other, and furthermore they may not be evaluated in quick succession, as described in section 4. As a result, the processing of a single rule can potentially be associated with at least four different database states: DB_T – the database at the start of the current transaction; DB_E – the database when the event took place; DB_C – the database when the condition is evaluated; DB_A the database when the action is executed. Active rule systems may support facilities within the condition of a rule that allow it to access zero or more of the states DB_T, DB_E and DB_C, and may also provide access to bindings associated with the event ($Bind_E$). The availability of information to the different components of a rule is illustrated in figure 3. In general, the position is even more complex than that portrayed in figure 3, as the state before and after an event has taken place may be different, and as multiple rules may be triggered and may execute to completion during the execution of a single action. As an example of the utility of such information, the

[1]Unlike the continuous context, an event occurrence does not participate in more than one composite computation in the cumulative context.

following rule is used to respond to the situation in which the `value` of the stock held by a `Holder` drops to `0`.

```
on update to value of Holder
if new.value = 0
do <action>
```

In this rule, information from the event (DB_E) is used to identify when the `value` field has been set to `0`, so that an appropriate response can be made (e.g. the `Holder` is deleted, information on the `Holder` is sent to the fund manager, etc). In other examples in this paper, conditions or actions access event parameters using `old` to refer to the value that a data item held before an event updated it, `insert` to refer to a newly inserted value, `delete` to refer to a recently deleted value, and `update` to refer to attributes of a data item that were unaffected by an update event.

3.3 Action

The range of tasks that can be performed by an action is specified as its **Options**. Actions may update the **structure** of the database or rule set, perform some **behavior invocation** within the database or an **external** call, **inform** the user or system administrator of some situation, **abort** a transaction, or take some alternative course of action using **do-instead** [Stonebraker et al. 1990]. As an example of *do-instead*, if an attempt was made to delete a tuple from the `Holder` relation that has a `value > 0`, then rather than allow the operation to proceed, the system manager could be informed of the attempted operation:

```
on delete to Holder
if delete.value > 0
do instead <inform system manager>
```

This is in contrast with the more standard semantics, in which the tuple is deleted *and* the system manager is informed:

```
on delete to Holder
if delete.value > 0
do <inform system manager>
```

The **Context** of the action is similar to that of the condition, and indicates the information that is available to the action, as illustrated in figure 3. It is sometimes possible for information to be passed from the condition of a rule to its action as DB_E or $Bind_C$. As an example of the utility of context information, the following rule is used to revise the data stored in the `value` attribute of all `Holder` tuples that are affected by a change in the `price` of some `Stock`:

```
on update to price of Stock
if true
do update Holder
   set value = value * (new.price/old.price)
   where reg# in (select reg# from Owns where stock# = update.stock#)
```

In this rule, both the `old` and the `new` values of the `price` have to be accessed (DB_E), as does the state of the database at the time of the update (DB_A).

Table 2. Dimensions for the execution model

Condition-Mode	$\subset$ {Immediate, Deferred, Detached}
Action-Mode	$\subset$ {Immediate, Deferred, Detached}
Transition Granularity	$\subset$ {Tuple, Set}
Net-effect policy	$\in$ {Yes, No}
Cycle policy	$\subset$ {Iterative, Recursive}
Priorities	$\in$ {Dynamic, Numerical, Relative, None }
Scheduling	$\in$ {All Parallel, All Sequential, Saturation, Some}
Error handling	$\subset$ {Abort, Ignore, Backtrack, Contingency}

4. EXECUTION MODEL

The execution model specifies how a set of rules is treated at run-time, and is characterized by the dimensions presented in Table 2 While the execution model of a rule system is closely related to aspects of the underlying DBMS (e.g. data model, transaction manager), there are a number of phases in rule evaluation, illustrated in figure 4, that transcend considerations that relate to specific software environments:

(1) The *signaling* phase refers to the appearance of an *event occurrence* caused by an event source.

(2) The *triggering* phase takes the events produced so far, and triggers the corresponding rules. The association of a rule with its event occurrence forms a *rule instantiation*.

(3) The *evaluation* phase evaluates the condition of the triggered rules. The *rule conflict set* is formed from all rule instantiations whose conditions are satisfied.

(4) The *scheduling* phase indicates how the rule conflict set is processed.

(5) The *execution* carries out the actions of the chosen rule instantiations. During action execution other events can in turn be signaled, that may produce *cascaded* rule firing.

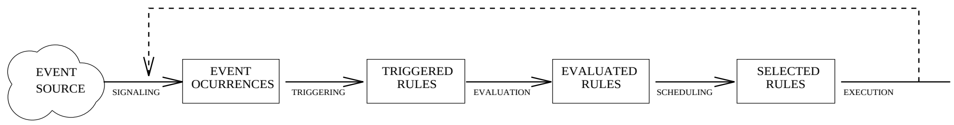

Fig. 4. Principal steps that take place during rule execution.

These phases are not necessarily executed contiguously, but depend on the **Event-condition** and **Condition-action** coupling modes. The former determines when the condition is evaluated relative to the event that triggers the rule. The **Condition-action** coupling mode indicates when the action is to be executed relative to the evaluation of the condition. The options for coupling modes most frequently supported are:

—**immediate**, in which case the condition (action) is evaluated (executed) immediately after the event (condition).

—**deferred**, in which case the condition (action) is evaluated (executed) within the same transaction as the event (condition) of the rule, but not necessarily at the earliest opportunity. Normally, further processing is left until the end of the transaction. However, some authors [Diaz and Jaime 1997] have also proposed to have a user-invoked coupling mode whereby the condition (action) is evaluated (executed) at a user-specified time after the event (condition) has been signaled (evaluated). A similar effect is also supported by Starburst [Widom and Finkelstein 1990] where users can invoke rule processing within a transaction by issuing special commands: the *process rules, process ruleset* S and *process rule* R commands invoke rule processing for the whole triggering rule set, a given subset S or a unique rule R, respectively.

—**detached**, in which case the condition (action) is evaluated (executed) within a different transaction from the event (condition). The execution of the action can be *dependent* upon or *independent* of the committing of the transaction in which the event took place or the condition was evaluated.

The nature of the relationship between events and the rules they trigger is partially captured by the **transition granularity**. This indicates whether the relationship between event occurrences and rule instantiations is 1:1 or many:1. When the **transition granularity** is **tuple**, a single event occurrence triggers a single rule. When the **transition granularity** is **set**, a collection of event occurrences are used together to trigger a rule. For example, if a rule R with a condition-action coupling mode of **deferred** is monitoring an event E, and occurrences e_1, e_2 and e_3 of E have taken place during a transaction, then the transition granularity indicates how many instantiations of R are created by the triggering phase. If the **transition granularity** is **tuple** then a separate instantiation of R is created for each of e_1, e_2 and e_3; if the **transition granularity** is **set** then a single instantiation of R is created to respond to the set of events $\{e_1, e_2, e_3\}$.

Another feature that influences the relationship between events and the rules they trigger is the **Net effect policy**, which indicates whether the net effect of the event occurrences rather than each individual event occurrence should be considered. The difference between the two strategies stems from cases in which several updates on the same data item can be considered as a single update: if an instance is updated and then deleted, the net effect is deletion of the original instance; if an instance is inserted and then updated, the net effect is the insertion of the updated instance; if an instance is inserted and then deleted, the net effect is no modification at all [Hanson 1992].

The question of what happens when events are signaled by the evaluation of the condition or action of a rule is addressed by the **Cycle policy** of the execution model. In general, there are two options. If the Cycle policy is **iterative**, then events signaled during condition and action evaluation are combined with those from the original event source illustrated in figure 4, and are subsequently consumed by rules from this single, global repository of signaled events. This means that condition or action evaluation is never suspended to allow responses to be made to events signaled by those conditions or actions. By contrast, if the Cycle policy is **recursive**, events signaled during condition and action evaluation cause the condition or action to be suspended, so that any immediate rules monitoring the events can be processed at the earliest opportunity. In practice, a recursive cycle policy is only likely to be considered in systems that support immediate rule processing, and some systems support a recursive cycle policy for immediate rules and an iterative cycle policy for deferred rules.

The **Scheduling** phase of rule evaluation determines what happens when multiple rules are triggered at the same time. The two principal issues are:

— *The selection of the next rule to be fired.* This topic has received much attention in the expert system community, as it is seen as fundamental to understanding and controlling the behavior of a set of rules [Winston 1984]. Indeed, rule order can strongly influence the result and reflects the kind of *reasoning* followed by the system. Examples of well-know **Dynamic** approaches (referred to as *conflict resolution policies*) are those that prioritize rules based on either the recency of update (i.e. the time of event occurrence) or the complexity of the condition. The former makes the system focus on a line of reasoning, since the most recently modified data is that associated with the most recently fired rule (ie. the search space is traversed depth-first). The latter reflects the assumption that condition complexity indicates the specificity of the rule (i.e. the extent to which the rule fits the current situation). However, mechanisms available in active database systems, that have to cope with large quantities of data efficiently in a context where deterministic behavior is held to be highly desirable, tend to support priority schemes in which rules are associated with a priority statically.
Static priorities are often determined either by the system (e.g. based on rule creation time) or by the user as an attribute of the rule. In the latter case, a rule is selected from a collection of simultaneously fired rules for execution using a **Priority** mechanism. Rules can be placed in order using a **numerical** scheme, in which each rule is given an absolute value that is its priority [Stonebraker et al. 1990], or by indicating the **relative** priorities of rules by stating explicitly that a given rule must be fired before another when both are triggered at the same time [Agrawal et al. 1991].

— *The number of rules to be fired.* Possible options include (1) To fire **all** rule instantiations **sequentially**. (2) To fire **all** rule instantiations in **parallel**. (3) To fire all instantiations of a specific rule before any other rules are considered, which is known as firing a rule to **saturation**. (4) To fire only one or **some** rule instantiation(s). Which approach is most appropriate depends upon the task that is being supported by the rule. The first alternative is suitable for rules supporting integrity maintenance: an update is successful once *all* constraints have been validated. The second option is described in HiPAC in a bid to encourage more efficient rule processing. The third option is most popular among expert-system practitioners, as it yields more focused inference than the other approaches. The fourth option can be of use to support derived data – different derivation criteria may be available (each of them supported by a rule), but only one is used.

A further aspect to be considered is how **Error handling** is supported during rule firing. Most systems simply **abort** the transaction, as this is standard behavior in databases. However, other alternatives may be more convenient [Hanson and Widom 1993]: to **ignore** the rule that raised the error and to continue processing other rules; to **backtrack** to the state when rule processing started and either restart rule processing or continue with the transaction; to adopt some **contingency** plan that endevours to recover from the error state, possibly using the

Table 3. Dimensions for rule management.

Description $\subset$ {Programming Language, Query Language, Objects}
Operations $\subset$ {Activate, Deactivate, Signal}
Adaptability $\in$ {Compile Time, Run Time}
Data Model $\in$ {Relational, Extended Relational, Deductive, Object-Oriented}
Programmer Support $\subset$ {Query,Trace}

exception mechanism of the underlying database system.

5. MANAGEMENT

Sections 3 and 4 have, respectively, described the structural characteristics of individual active rules and the run-time evaluation of sets of such rules. This section considers the facilities provided by the system for managing rules – specifically what operations can be applied to rules, how rules are themselves represented, and programming support for rules. Possible dimensions are shown in Table 3.

The **Description** of rules refers to how rules themselves are normally expressed using either a database **programming language** [Gehani et al. 1992; Buchmann et al. 1995], a **query language** [Widom and Finkelstein 1990; Stonebraker et al. 1990], or as **objects** in an object-oriented database [Dayal et al. 1988; Diaz et al. 1991]. These categories are not exclusive. For example, it is possible for an extended query language facility in an object-oriented database to arrange for rules to be stored as objects.

Besides creation and deletion, which are taken to be mandatory, other **Operations** commonly found are **activate, deactivate** and **signal**. Activation (deactivation) of rules makes the system start (stop) monitoring the rule's event or condition. Since rules can persist for long periods, this mechanism helps the database administrator to temporarily switch on (off) some rules without deleting them. Among other things, such deactivation mechanisms may be convenient for improving efficiency, for debugging, or for loop prevention (e.g. by deactivating rules once they have been fired). This mechanism may be available for individual rules as well as for rule sets. The latter can help in structuring the rule base, as well as speeding up rule processing.

The operation **Signal** is required to support *abstract events*, and is invoked explicitly by the application to notify the rule system of external occurrences.

Although all active DBMSs support creation and deletion of rules, they can differ in the level of **Adaptability** supported. In some systems it is only possible to change the rules associated with an application by recompiling the application code, and thus the rules can only be modified at **compile time**. Others support more dynamic **run time** rule modification, including the ability of rule actions to modify the rule base. Clearly there is a sliding scale of degrees of **Adaptability** – in the context of the dimensions, any system that allows rules to be created without recompiling application code can be considered to support **run time** adaptability.

There is an extent to which the **Data Model** with which an active rule system is associated is independent of the other dimensions of rule system functionality. However, the data model is likely to significantly influence the designers of an active rule system, and is thus included as a dimension.

Programmer Support is of paramount importance if active rules are to be adopted as a mainstream implementation technology in business environments. Important facilities for supporting the developers of rule bases include the ability to **query** the rule base and to **trace** or in some other way monitor rule system behavior. Support for application development is discussed more fully in section 8.

6. ACTIVE RULE SYSTEMS

The previous three sections have introduced many of the principal features of active database systems, and together constitute a framework within which active functionality can be described. In this section the framework is applied to the presentation of a range of prominent proposals for active database systems, thereby highlighting important similarities and differences.

6.1 Relational Systems

The inclusion of active behavior modeling facilities in relational databases is not particularly new, and most commercial systems include triggering mechanisms. In addition, a number of research prototypes have been developed that seek to provide more comprehensive support for active rules. Proposals for including active behavior in relational systems often have a range of common characteristics, which stem from the nature of the underlying passive database system. For example, rules are generally triggered by system defined operations on the structure of the database (e.g. insert a tuple, modify a tuple), because until recently, relational systems have rarely supported user-defined operations. It can be anticipated that future active rule systems for relational databases will be extended so that primitive events include the execution of stored procedures, but such functionality is not supported by the systems reviewed here. Furthermore, because relational languages such as SQL provide facilities for expressing conditions and for performing updates, the rule description language is often an extension of the query language. In general, active mechanisms proposed for relational databases support only one or a limited range of coupling modes, and have limited support for composite event detection. This, however, is not because of any fundamental restrictions imposed by the relational model, and extended proposals may emerge in due course.

Table 4 indicates how four proposals for the inclusion of active behavior into relational systems, namely Starburst, POSTGRES, Ariel and SQL-3, fit into the framework introduced in the previous three sections. The following subsections describe distinctive features of these proposals. Other examples of active extensions to systems with largely relational data models are presented in [Kotz et al. 1988; Kiernan et al. 1990; Zaniolo 1994; Harrison and Dietrich 1994; Bayer and Jonker 1994; Reddi et al. 1995].

6.1.1 *Starburst.* The Starburst active rule system adds active functionality to an extensible relational database system [Widom and Finkelstein 1990], and has been used as a testbed for a number of database internal applications, including integrity constraints [Ceri et al. 1990] and materialized views [Ceri and Widom 1991].

Perhaps the most noteworthy feature of the Starburst rule system is its set-based execution model, in which rules are triggered by the net effect of a set of changes to the data stored in the database. When an operation takes place that is being monitored by a rule, the nature of the change is logged in a transition table. Entries

Table 4. Dimensions applied to active relational database systems.

Dimension	**Starburst**	**POSTGRES**	**Ariel**	**SQL-3**
Knowledge Model				
EVENT:				
Source	Structure Op	Structure Op	Structure Op	Structure Op
Granularity	Set	Set	Set	Member Subset, Set
Operators	or			
Cons'n. Policy	Cumulative	N/A	N/A	N/A
Role	Mandatory	Mandatory	Optional	Mandatory
CONDITION:				
Role	Optional	Optional	Optional	Optional
Context	Bind_E,DB_C	Bind_E,DB_C	Bind_E,DB_C	Bind_E,DB_E DB_C
ACTION:				
Options	Structure Op. Abort Update-Rules	Structure Op. Abort Do Instead	Structure Op.	Structure Op. Behavior Inv. External Do Instead
Context	Bind_E, Bind_C,DB_A	Bind_E,DB_A	Bind_E,DB_A	Bind_E,DB_E DB_A
Execution Model				
Condition-Mode	Deferred	Immediate	Immediate Deferred	Immediate
Action-Mode	Immediate	Immediate	Immediate	Immediate Deferred
Trans. Gran.	Set	Tuple	Set	Tuple, Set
Net-effect policy	Yes	No	Yes	No
Cycle Policy	Iterative	Recursive	Iterative	Recursive
Priorities	Relative	None	Numerical	None
Scheduling	All Sequential	All Sequential	All Sequential	All Sequential
Error handling	Abort	Abort Exception	Abort	Backtrack
Management				
Description	Query Lang.	Query Lang.	Query Lang.	Query Lang.
Operations	Signal, Activate Deactivate			
Adaptability	Run Time	Run Time	Compile Time	Run Time
Prog. Support				

in such tables can then be revised to take account of subsequent update operations – for example, if a tuple is inserted and then updated, the *net-effect* is logged as an insert of the updated tuple; if a tuple is inserted and then deleted, the *net-effect* is that no operation has taken place. The information that is stored in transition tables is used to trigger rules at *rule assertion points*, that may take place either during or at the end of a transaction. In this context, events do not trigger rules directly. Rather, the fact that an event has occurred is recorded in a transition table for subsequent consideration, and the timing or order of specific events is not taken into account by the scheduler.

Each rule that is monitoring a particular event has access to the *net-effect* of all updates of relevance to that event, and that have not already been considered by the rule – when a rule is fired multiple times within a single transaction, the changes it has access to are those that have taken place since the last firing of the rule.

The Starburst rule system can be considered to be conservative in its design, in that a modest and fixed set of facilities are supported. However, the semantics of rule execution in Starburst are still quite complex, and it can be argued that supporting many more facilities is likely to lead to rule sets that are difficult to understand and maintain.

6.1.2 *POSTGRES.* The active rule system is only one of a number of extensions to the relational model supported within POSTGRES [Stonebraker and Kemnitz 1991], with others facilitating the representation of objects and user-defined operations. The POSTGRES rule system is considered within the section on relational databases because the object-oriented features of POSTGRES seem to have had little bearing on the design of the rule system. The POSTGRES rule system, like that of Starburst, can be considered to be quite conservative, although a key distinction is that the POSTGRES rule system is tuple-oriented. Furthermore, the consequences of an event being raised take effect immediately, rather than being deferred until a later rule assertion point.

6.1.3 *Ariel.* An important characteristic of Ariel that distinguishes it from Starburst or POSTGRES is the optionality of the event – Ariel principally supports *condition-action* rules. The consequences of this distinction for the implementation of rule systems is considered in section 7.5.

The suitability of condition-action rules compared with ECA-rules depends upon the context. There are circumstances in which it is necessary to make the event part of a rule explicit to capture the semantics of an application. For example, in the example application from section 2, there might be a general policy that no more than 10% of the total value of the stock owned by *Cautious Investments* should be of the one kind. This could be captured by a condition-action rule thus:

```
if h.name = "Cautious Investments" and h.reg# = o.reg# and
   o.stock# = s.stock# and o.qty * s.price > 0.1 * h.value
   from h in Holder, o in Owns, s in Stock
do <reduce quantity of s owned by h>
```

However, this rule would have the effect of selling a particular kind of stock without regard for the reason why its value had increased relative to the total value held by *Cautious Investments*. It may be preferable to distinguish between the different events that caused the condition to go true using ECA-rules. For example, if the condition became true as a result of the purchase of more of the stock, then the update may be blocked. By contrast, if the condition became true as a result of an increase in the price of the stock, then the portfolio manager could be warned, but no action taken. These approaches can be supported by the following rules:

```
on update to qty of Owns
if h.name = "Cautious Investments" and
   h.reg# = new.reg# and
   new.stock# = s.stock# and
   new.qty * s.price > 0.1 * h.value
   from h in Holder, s in Stock
do abort

on update to price of Stock
if h.name = "Cautious Investments" and
   h.reg# = o.reg# and
   o.stock# = new.stock# and
   o.qty * new.price > 0.1 * h.value
   from h in Holder, o in Owns
do <inform portfolio manager>
```

Thus ECA-rules can be considered to describe the context of a rule in a more *precise* way than condition-action rules. This may not, however, always be to the advantage of programmers using a rule system. For example, to describe the situation captured by the above condition-action rule using ECA-rules, events would have to be defined that monitor the insertion of tuples into the `Owns` table, changes to the `reg#` attribute of `Owns` and `Holder`, updates to the `value` attribute of `Holder`, etc. Thus, by providing both ECA-rules and condition-action rules, Ariel can be said to offer the best of both worlds in contexts such as the above.

6.1.4 *SQL-3.* Most commercial relational databases support trigger mechanisms. However, the knowledge and execution models of these mechanisms have traditionally differed from system to system. With a view to providing more consistent support for active mechanisms in relational systems, triggers are included in the emerging SQL-3 standard [Kulkarni et al. 1998]. This survey features the standard rather than any of the current commercial systems, as it is expected that the commercial systems will drift towards the standard in due course.

The SQL-3 standard, like many commercial systems, supports both row level triggers (with a transition granularity of tuple) and statement level triggers (with a transition granularity of set). Statement level triggers are executed once in response to an update operation on a table, no matter how many tuples are affected by the update. However, perhaps the most important feature of the SQL-3 standard is that it makes explicit how triggers are to interact with other features found in relational databases, and in particular, declarative integrity checking mechanisms.

6.2 Object-Oriented Systems

Object-oriented databases (OODBs), unlike their relational predecessors, have always supported a close association of user-defined behavior with database data. Such behavior is generally expressed as methods attached to the classes that structure the data stored in the database. This, plus the hiding of certain aspects of the structure of an object using encapsulation, means that certain of the tasks that may be performed by active behavior in relational databases are supportable using method code in object-oriented systems. Despite this, proposals for active extensions to OODBs abound, with early proposals being made only a few years after the first work on passive OODBs. This rapid and extensive research activity has probably been encouraged by the tendency for OODBs to be used in advanced applications, where the need for comprehensive behavior management facilities is greater than in more traditional domains. Furthermore, the nature of certain applications has encouraged investigation of active database constructs that are significantly more powerful than those described for relational systems in figure Table 4, as shown for selected systems in Tables 5 and 6. As well as being more powerful than most extensions to relational databases, a common difference is that primitive events in active OODBs are often associated with method invocations rather than access to or update of structure. This partly derives from the desire to avoid reducing data independence by linking active behavior to structure directly, and partly from the fact that some systems use layered architectures in which it is not straightforward to raise events on the basis of structure operations.

The following subsections outline the principal features of eight projects developing active object-oriented databases, four of which are not based upon persistent C++ systems (HiPAC, EXACT, NAOS and Chimera, as featured in Table 5), and four of which are based on database extensions of C++ (Ode, SAMOS, Sentinel and REACH, as featured in figure Table 6). Other examples of active extensions to OODBs are presented in [Dittrich 1993; Etzion et al. 1994; Naqvi and Ibraham 1994; Thomas and Jones 1995; Dinn et al. 1996].

6.2.1 *HiPAC.* The HiPAC project [Dayal et al. 1988; Chakravarthy 1989; Dayal 1989] pioneered many of the most important ideas in active databases, such as coupling modes and composite events, although the resulting design was not fully implemented. HiPAC was associated with the passive OODB PROBE, and objects in this model were used to store the ECA-rules of the active extension. Further distinctive features of HiPAC are the parallel execution of triggered rules as subtransactions, the extension of the query algebra with a *changes* operator that allows access to *delta* relations that monitor the net effect of a set of changes, and identification of real-time applications that can benefit from active database facilities.

6.2.2 *EXACT.* EXACT, an EXtensible ACTive rule manager [Diaz et al. 1991; Diaz and Jaime 1997], adds active facilities to the OODB ADAM, in which instances, classes, rules and events are represented uniformly as database objects. Two contentions support this work, specifically: that control information rarely refers to single rules but to sets of rules, and that rules supporting different applications often require different execution models. To support these contentions, EXACT provides an extensible rule model in which collections of rules can be de-

Table 5. Dimensions applied to active object-oriented systems.

Dimension	**HiPAC**	**EXACT**	**NAOS**	**Chimera**
Knowledge Model				
EVENT:				
Source	Structure Op. Behavior Inv. Transaction Clock Abstract	Behavior Inv. Transaction Exception Clock, External Abstract	Structure Op. Behavior Inv. Transaction Clock, Program Abstract	Structure Op. Behavior Inv.
Granularity	Member,Set	Member,Set	Set	Set
Operators	or, seq, times	and, or	and, or, not seq	or
Cons'n policy		Cumulative		Cumulative
Role	Mandatory	Mandatory	Mandatory	Mandatory
CONDITION:				
Role	Optional	Mandatory	Optional	Mandatory
Context	Bind$_E$, DB$_C$	Bind$_E$, DB$_C$	Bind$_E$, DB$_C$	Bind$_E$, DB$_T$ DB$_C$
ACTION:				
Options	Structure Op. Behavior Inv. Update Rules Abort External	Behavior Inv. Update Rules Abort External	Structure Op. Behavior Inv. Update Rules Abort, External Do Instead	Structure Op. Behavior Inv. Abort
Context	Bind$_E$, Bind$_C$ DB$_A$	Bind$_E$, Bind$_C$, DB$_A$	Bind$_E$, Bind$_C$ DB$_A$	Bind$_E$, Bind$_C$ DB$_T$, DB$_A$
Execution Model				
Condition-Mode	Immediate Deferred Detached	Immediate Deferred	Immediate Deferred	Immediate Deferred
Action-Mode	Immediate Deferred Detached	Immediate Deferred	Immediate	Immediate
Trans. Gran.	Set	Tuple	Tuple, Set	Set
Priorities	Relative	Relative Numerical	Relative	Relative
Net-effect policy	Yes	No	Yes	Yes
Cycle Policy	Iterative Recursive	Recursive	Iterative Recursive	Iterative
Scheduling	Parallel	All Sequential Some	All Sequential	All Sequential
Error handling	Contingency	Abort, Ignore	Abort	Abort
Management				
Description	Objects	Objects	Query Lang. Prog.Lang.	Query Lang.
Operations	Activate Deactivate Signal	Activate Deactivate Signal	Activate Deactivate Rename	Activate Deactivate
Adaptability	Run-time	Run-time	Run-Time	Compile Time
Prog. Support	Query	Query, Trace	Query	Query, Trace

Table 6. Dimensions applied to active object-oriented systems based on C++.

Dimension	**Ode**	**SAMOS**	**Sentinel**	**REACH**
Knowledge Model				
EVENT:				
Source	Behavior Inv.	Behavior Inv. Transaction Clock, Abstract	Behavior Inv. Transaction Clock	Structure Op. Behavior Inv. Transaction Clock
Granularity	Member,Set	Member,Set	Member,Set	Set
Operators	and, or, not seq, times	and, or, not seq, times	and, or, seq, times, any	and, or, seq, times, not
Cons'n policy	N/A	Chronicle	Recent Chronicle Continuous Cumulative	Recent Chronicle
Role	None	Mandatory	Mandatory	Mandatory
CONDITION:				
Role	Mandatory	Mandatory	Optional	Mandatory
Context	DB_C	$Bind_E$,DB_C	$Bind_E$,DB_C	$Bind_C$,DB_C
ACTION:				
Options	Behavior Inv.	Structure Op. Behavior Inv. Inform, External, Abort	Structure Op. Update-Rules Inform Abort	Structure Op. Behavior Inv. Inform External, Abort
Context	DB_A	$Bind_E$,DB_A $Bind_C$	$Bind_E$,DB_A	$Bind_E$,$Bind_C$ DB_A
Execution Model				
Condition-Mode	Immediate	Immediate Deferred Detached	Immediate Deferred	Immediate Deferred Detached
Action-Mode	Immediate Deferred Detached	Immediate Deferred Detached	Immediate	Immediate
Trans. Gran.	Tuple,Set	Tuple	Tuple	Tuple
Priorities	none	Relative	Relative	Numerical
Cycle Policy	Iterative	Recursive	Recursive	Iterative Recursive
Scheduling	All Sequential	All Sequential	Parallel	Parallel
Net-effect policy	No	No	No	No
Error handling		Abort	Abort	Contingency
Management				
Description	Prog. Lang.	Objects	Objects	Prog. Lang Objects
Operations	Activate Deactivate	Activate Deactivate	Activate Deactivate	Activate Deactivate
Adaptability	Compile-time	Run-time	Run-time	Compile-time
Prog. Support		Query, Analyse	Query, Trace	Query, Trace

veloped that share similar features, the functionalities of which are described by specializing the general rule management facilities provided with the system. EXACT has been used for experimentation in the development of advanced database facilities [Diaz 1992; Paton et al. 1993; Diaz et al. 1994].

6.2.3 *NAOS.* NAOS [Collet et al. 1994] is an active rule system for the O_2 commercial OODB [Deux and et al. 1990]. As NAOS has been implemented as part of the kernel of O_2, rather than as a layer on top that has not been sanctioned by the vendor, it may develop into the first commercially available active OODB.

As O_2 provides comprehensive support for two languages, O_2C and OQL, NAOS has been able to exploit this to provide declarative expression of conditions using OQL and powerful action expression using O_2C. The execution model of NAOS is slightly unusual, in supporting depth-first, recursive processing of immediate rules, but breadth-first, iterative processing of deferred rules. The NAOS rule system has been formally specified using denotational semantics [Coupaye and Collet 1995], and is also being used for experimental work on optimization [Collet and Manchado 1995].

6.2.4 *Chimera.* The Chimera [Ceri et al. 1996] active rule system is unique among those surveyed here in building upon a deductive object-oriented database (another such system is described in [Dinn et al. 1996]). The use of the deductive language for condition expression encourages a set-oriented view of rule processing, and information is passed from the event to the condition by querying an event history. Another unusual feature of Chimera is that rule conditions and actions can access past database states, either at the start of the current transaction, or when the rule was last fired. Chimera is implemented by compiling user statements into an internal form that is interpreted by a run-time system that stores both database and rule system data using the ALGRES extended relational storage manager.

6.2.5 *Ode.* The Ode database system is defined and manipulated using the O++ database programming language, which extends C++ with database facilities (e.g. persistence, versions). It provides two categories of rule that are semantically divided into *constraints* and *triggers*, each with a different syntax and execution model [Gehani and Jagadish 1991]. Although triggers can, in general, be used to support constraints, the authors argue that the distinction clarifies the role that is being played, and facilitates more efficient implementation. Both constraints and triggers are defined at the class level, and are subject to inheritance like other properties of a class.

A constraint consists of a predicate on the state of an object and a single action to be executed if the condition becomes false. Once the action has been executed, the system checks the condition again. If it is still false, the transaction is aborted. As for the event-condition coupling modes, different options are supported depending on whether constraints should be checked immediately (declared as *hard constraints*), or deferred until the end of transaction (declared as *soft constraints*). Multiple updates affecting the same hard constraint in the course of a transaction cause the constraint to be evaluated once after each update, whereas for a soft constraint the check is only carried out once at the end of the transaction. Constraints affect all instances of a class, and are permanently activated.

Unlike constraints, triggers have to be explicitly activated on particular objects. If a trigger is active, then when its condition becomes true, its action is executed. Once signaled, triggers declared as *once-only* are automatically deactivated whereas those declared as *perpetual* are reactivated automatically. The user can explicitly deactivate a trigger using the command *deactivate*. Unlike constraints, triggers with a condition that has evaluated to true are executed in a separate transaction after the current transaction has committed (i.e. the event-condition coupling mode is immediate, whereas the condition-action coupling mode is detached dependent). Note that the processing supported by triggers in Ode is quite distinctive, and open to interpretation in different ways. Here, triggers have been considered as condition-action rules, but an equally valid interpretation is that they are event-action rules in which the event is defined using a rich event algebra and boolean expressions, known as masks.

6.2.6 *SAMOS.* SAMOS [Gatziu et al. 1991; Gatziu and Dittrich 1994] provides active facilities on top of the ObjectStore commercial OODB. As the developers of SAMOS did not have access to the source of ObjectStore, the active system is implemented as a layer on top of ObjectStore, rather than as part of the kernel.

Perhaps the most significant feature of SAMOS is its event detector, the semantics of which is based upon petri nets, and which, in turn, is implemented using a graph structure that reflects the structure of the petri net. The event language itself presents users with a series of operators that are shared by other systems with comprehensive event languages, such as HiPAC [Dayal et al. 1988], Sentinel [Chakravarthy et al. 1994] or REACH [Buchmann et al. 1995].

6.2.7 *Sentinel.* Sentinel [Chakravarthy et al. 1994; Chakravarthy et al. 1994] is an active extension to the C++ based OpenOODB system from Texas Instruments [Wells et al. 1992]. The focus in this project has been upon the provision of comprehensive event specification mechanisms, representation of rules as database objects, and integration of the rule system with a sophisticated transaction manager. In particular, the consumption modes that are now widely accepted as providing appropriate descriptions of how to construct parameters to composite events in rule systems with tuple level transition granularities were first implemented in Sentinel.

6.2.8 *REACH.* REACH [Buchmann et al. 1995] has much in common with Sentinel, in that it too is an active extension to OpenOODB. The emphasis in the REACH project has been on developing a comprehensive understanding of how the rule manager interacts with complex transaction models, with a view to supporting open applications [Branding et al. 1994]. Examples of coupling modes supported in REACH with open applications in mind are *sequential causally dependent*, in which a rule executes in a separate transaction from its triggering event only after the triggering transaction commits, and *exclusive causally dependent*, in which a rule executes in a separate transaction from its triggering event only after the triggering transaction aborts.

7. ARCHITECTURAL ISSUES

This section considers how some of the characteristics of active database systems presented in the above sections can be implemented. In addressing certain issues, reference is made to the abstract architecture of an active database system presented in figure 5. This figure makes explicit the principal processes (rectangles) and data stores (ellipses) used to implement the functionality illustrated in figure 4.

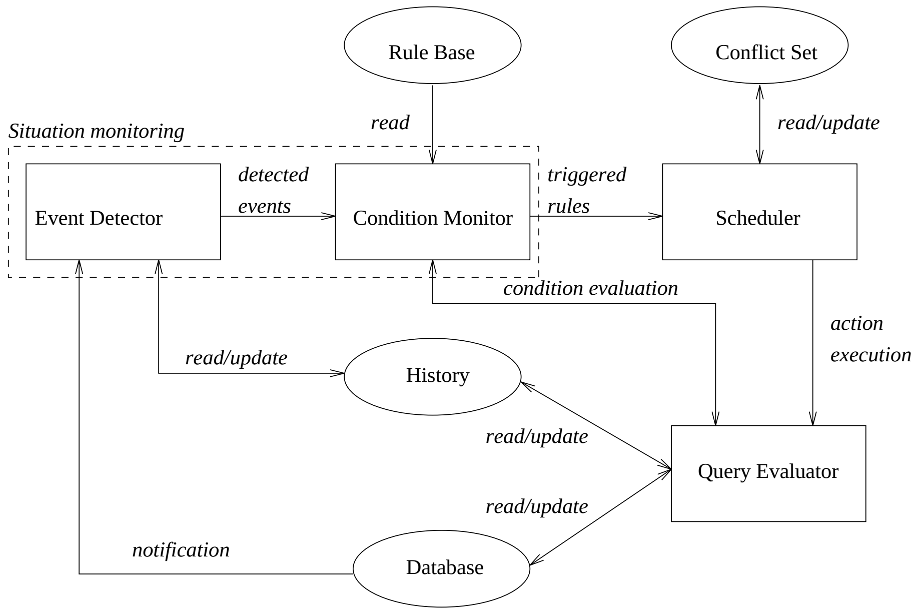

Fig. 5. Abstract active rule system architecture

The principal processes are as follows:

(1) The **Event Detector**, which ascertains what events of interest to the rule system have taken place, if any. Primitive events are notified from the database or from external sources; composite events are constructed from incoming primitive events plus information about past events that can be obtained from the history.

(2) The **Condition Monitor**, which evaluates the conditions of rules associated with events that have been detected by (1). In systems that support condition-action rules, there is no explicit statement of the events to be monitored, although actual implementations do have to monitor primitive events. The distinctive implementation strategies that are often used in such systems are considered in section 7.5.

(3) The **Scheduler**, which compares recently triggered rules with those that have previously been triggered, updates the conflict set, and fires any rules that are scheduled for immediate processing.

(4) The **Query Evaluator**, which executes database queries or actions. Access may be required both to the current state of the database and to past states, in order to support monitoring of how the database is evolving.

The functionality of each of the above components depends very much on the knowledge and execution models of the active database system to be supported, which in turn are influenced by the environment within which the active database is being developed. Architecturally, two principal categories of active database can be identified:

Layered:. The active component is developed as a layer of software on top of an unchanged passive database system. This approach has the advantage that no access is required to the source code of the passive database system, and that the resulting active system may be easily portable for use with different passive systems. However, the lack of access to the kernel of the underlying database may impact upon performance and limit what functionality can be supported in terms of primitive event detection, coupling modes and optimization.

Integrated:. The active component is developed by changing the source of an existing passive database system. This approach frees the designer of the active database system from the limitations of the layered approach, and is probably the preferred model for developing industrial strength systems. It is worth noting, however, that the number of changes required to the kernel of a system to allow it to support active capabilities effectively is often not large, and that practical systems can be developed using largely layered software, with a small number of hooks into the kernel.

The first systematic performance evaluation of different active database systems and architectures is provided by [Geppert et al. 1996]. The following subsections address a range of implementation issues in more detail.

7.1 Event Detection

There are two principal aspects to the implementation of event detection – the monitoring of primitive events and the accumulation of information that is of relevance to composite events.

The detection of primitive events normally involves some form of check within the kernel of the database system, a characteristic that means that it is often not possible for active functionality to be implemented as a layer on top of an existing database system. For example, to detect that an update has been made to a tuple in a relation, it is necessary for the update operation of the database system to be able to identify which primitive events are associated with the specific update being performed. As a further example, in object-oriented databases events are often raised in response to the invocation of user defined methods, in which case the event detector must be notified of message-sending events by the method dispatcher. Specific techniques for detecting primitive events in object-oriented systems are discussed in [Dittrich 1993; Gatziu and Dittrich 1994; Chakravarthy et al. 1994; Kim et al. 1992]. Broadly speaking, this can be achieved using a sender-based, a receiver-based or a dispatcher-based mechanism [Fernandez and Diaz 1995]. The former does not really lead to active systems, as event detection is undertaken by the application rather than the DBMS itself. The application is changed wherever the relevant message is sent. As well as being intrusive, the main drawback of this approach is that detection is replicated, distributed and embedded in application programs, thus jeopardizing encapsulation and maintenance.

The receiver-based approach is mainly based on *wrappers*: a monitored message must activate an operation that extends the original code provided by the user with the associated signal to the event detector. This extra code is called a wrapper since it is typically executed before and/or after the original method. The receiver-based approach is generally found in layered architectures, as primitive event detection can be implemented by preprocessing application programs, and thus changes to the kernel of the database are not required.

By contrast, dispatcher-based mechanisms offer a general solution by placing the signaling code inside the database system itself. Such mechanisms can be used to monitor a range of aspects of system behavior, including transaction operations, structural primitives and behavior invocations. As this approach requires changes to the kernel of the database, it is found only in systems with integrated architectures.

As for composite event detection, it involves recording information on all partially detected composite events that may become fully detected in the future. For example, in the composite event $E_1 and E_2$, if E_1 has taken place but E_2 has not, then this fact must be recorded until E_2 either does take place or the timespan within which the event is to be detected expires. For composite events whose components are primitive events that have originated within the boundaries of a single transaction, the end of the transaction marks the end of the monitoring period, and all partially composed events can be removed. If the source transaction is not relevant, so that a composite event can be formed by events that originate in different transactions, a validity interval is required. This may be given either for the whole composite event, or it may be determined by the smallest validity interval of the composing events [Buchmann et al. 1995].

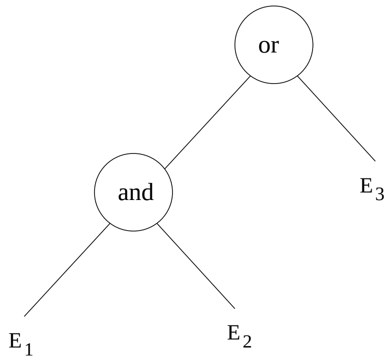

Fig. 6. Example of Sentinel event graph.

A range of different structures have been proposed for implementing composite events along with event algebras. For example, [Gatziu and Dittrich 1994] describes the use of colored Petri-nets for specifying and implementing an event detector, whereas [Gehani et al. 1992] uses finite state automata, and [Chakravarthy et al. 1994] uses an event graph that is linked directly to the query graph used to express the condition of the corresponding rule. As an example of how information on partially detected events is accumulated, figure 6 shows the event graph that would be used to describe the event E_1 *and* E_2 *or* E_3. As events take place, the event graph is decorated with instances of primitive events that are then consumed as composite events are detected. For the example, in figure 6, in all event consumption modes except **Recent** mentioned in section 4, if N occurrences of E_1 are raised before any occurrences of E_2, then all N occurrences of E_1 must be stored in anticipation of the subsequent raising of some matching occurrences of E_2. Such a process can be very expensive, and in practice care must be taken in the design and use of composite event detectors to ensure that the history data-store in figure 5 does not become extremely large.

7.2 Condition Monitoring

The event combined with the condition describes the *situation* that an ECA-rule is monitoring. As such, there is often a close association between the event detector and the condition monitor – the event detector initiates processing within the condition monitor, and information about the events that have occurred must be passed from the event detector to the condition monitor. The more sophisticated the event detector, the more complex will be the information to be passed to the condition monitor. For example, in the case of a primitive event such as `on insert to Stock`, the only information that needs be passed to the condition monitor about the event is the inserted tuple. By contrast, where the event is composite, the relevant information about the event is also more complex. For example, in a conjunctive event, information should be available to the condition of the rule on all of the events that caused the conjunctive event to be signaled.

Once rules that are associated with detected events have been retrieved from the rule base and bound to the parameters of these events, they must be passed to the query evaluator to establish which rules have satisfied conditions. The query processor is likely to be an extension of that used with a passive database, as rule conditions are parameterized with bindings from events, and may also have access to past states of the database. For example, in Starburst [Widom et al. 1991] a *transition log* records how tables have changed during a transaction, which supports access from the condition of the rule to the accumulated changes associated with the event that has triggered the rule. A comparable mechanism is required by any system that supports rule processing based upon the net effect of a set of accumulated changes. The contents of the history in a specific rule system thus depend upon the nature of the event specification language supported, plus the scope of access to past database states from the condition/action of the rule.

Some condition languages are based on the declarative query language of the underlying database, which raises the possibility of applying optimization techniques to rule conditions. In general, the existing optimizer can be applied directly to the optimization of such conditions, and significant gains in performance are likely to be experienced where event parameters are used to restrict the information accessed by the condition. A similar theme is explored by [Baralis and Widom 1995], where it is shown how exploitation of intermediate information held by the rule manager can be used to speed up condition evaluation, although this approach only provides significant gains where there is repeated triggering of individual rules. Further, in systems with rich event algebras that are able to match specific values in events, an optimizer may be able to move some of the selections from the condition into the event detector. It is also shown in [Collet and Manchado 1995] how detailed analysis of rule conditions and actions can be used to identify rules that are amenable to parallel execution.

The design and implementation of effective situation monitors for active database systems is of considerable importance, and various balances have to be struck between the provision of expressive facilities and efficiency of processing. For example, expressive event algebras enable more of the situation that is being monitored to be described within the event part of the rule, which in turn leads to less frequent invocation of simplified rule conditions. However, the reduced cost of condition evaluation must be balanced against the considerable overheads associated with composite event detection. A definitive position on the trade-offs involved in situation monitoring awaits further theoretical and empirical analyses of alternative approaches.

7.3 Action Execution

Rules with conditions that are satisfied are prepared for execution by the scheduler, which also maintains the conflict set of triggered rules that have yet to be fired. The complexity of the scheduler also varies considerably from system to system – for example, in POSTGRES [Stonebraker et al. 1990] the scheduler is quite straightforward, as all rules are fired as soon as they are triggered, and because there is no priority scheme that requires rules to be fired in a specific order. By contrast, in Starburst [Widom et al. 1991] the scheduler is more complex – rules must be fired in an order that is consistent with a priority graph, and it is possible for a rule to be removed from the conflict set without being fired because the processing of rules is based upon the *net effect* of the changes to the underlying database.

When a rule is scheduled for execution, its action is passed to the query evaluator, which in turn is likely to update the database and the history. The action of a rule is also likely to have access to information on the event that caused the rule to be triggered (i.e. $Bind_E$), and may also be passed data that has been retrieved by the condition of the rule (i.e. $Bind_C$) (e.g. the set of objects for which an integrity constraint has been violated).

Binding can be supported in different ways. In HiPAC and EXACT, explicitly specified parameters are available to provide access to the current event. For example, the predicate *current_occurrence(O)* can be used in rule conditions or actions in EXACT to obtain the parameters of the event occurrence that has triggered the rule. Starburst stores binding information in transition tables that record the net effect of tuple modifications caused during query processing. Transition tables can be queried by the condition or the action of a rule. Unlike Starburst, POSTGRES has tuple-based processing, which considerably simplifies the bindings, since only the current tuple needs to be considered. Correlation names *new* and *old* are provided to allow access the value of the current tuple before and after modification.

7.4 Transaction Management

It is often the case that sophisticated facilities in the associated passive database system can be used to increase the functionality of the active mechanisms. For example, in active object-oriented systems much has been made of the ability to associate rules with user-defined operations, and to share active behavior within inheritance hierarchies. Most implemented active database systems are associated with conventional flat transaction models, and thus rules are processed to completion within the same transaction as the events that led to their triggering. However, where more sophisticated transaction management facilities are available, these can be exploited to increase the flexibility of conventional rule systems, and to allow addition of features that can be used to support more advanced applications.

Rich transaction models can be used to underpin a range of different behavioral extensions to execution models [Beeri and Milo 1991]. A *nested transaction* is a transaction that contains within it a number of component transactions, or *subtransactions* [Moss 1985]. The nested transaction creates the subtransitions and waits until they terminate. This model can be extended so that the nested transaction can run concurrently with the subtransactions [Harder and Rothermel 1993]. As well as speeding up the whole process as a result of increased concurrency, this model can be useful for active database systems. For example, if the action of a rule is unable to carry out the task assigned to it, then there is a significant chance that the whole transaction will be aborted, potentially undoing a significant amount of work. By contrast, if the action of the rule is executed in a separate subtransaction (the *triggered transaction*) which aborts, then the parent transaction (the *triggering transaction*) is free to decide how to respond to the failure – it could repeat the action a set number of times, fire an alternative action, etc. Such underlying functionality can be made available to the rule programmer as extensions to the rule definition that describe how to respond to failures.

In addition, a number of issues can be identified that relate to the relationship between concurrency control and coupling modes. If the triggering and triggered transaction are executed concurrently, it could happen, depending on the concurrency control method used, that the triggered transaction commits while the triggering transaction is still executing (and thus, potentially aborting). This possible behavior jeopardizes the concept of causality: an effect should not precede its cause [Hsu et al. 1988]. To fall into line with this concept, a schedule must obey the following two rules: the triggered transaction must be serialized after the triggering transaction, and the triggered transaction can commit only if the triggering transaction commits. The user may either choose to obey the causality principle or to allow the triggered transaction to be executed freely. Using the terminology introduced in section 4, these options correspond to the **detached dependent** and **detached independent** coupling modes, respectively.

7.5 Production Rule Algorithms

The architecture presented in figure 5 makes explicit the role of event detection for situation monitoring, whereas some systems, such as Ariel [Hanson 1992] and Amos [Skold and Risch 1995] support production (or condition-action) rules without explicit event specifications. In practice, however, primitive update events must also be detected by database production systems, as the truth of a condition can be changed as a consequences of changes to the underlying database. Figure 5 can thus be seen as a potential architecture for implementing a production rule system, although algorithms commonly used for improving the efficiency of condition evaluation are quite different from those generally applied in the context of ECA-rules.

Condition-action rules are processed in the context of the following recognize-act cycle:

```
match
while (conflict set not empty) do
    conflict resolution
    act
    match
end-while
```

In such a cycle, the `match` phase identifies rules with conditions that are true with respect to the state of the database, and adds them to the `conflict set`, which is essentially a queue of triggered rules waiting to be fired. The `conflict resolution` step selects a single rule from the conflict set for further processing in the `act` phase, which executes the statements in the action of the selected rule.

A naive evaluation of the `match` phase would evaluate the condition of every rule to find which rules should be considered for processing. Such an approach would be prohibitively expensive, and is not necessary, as only a small part of the database over which each condition acts is likely to change during each cycle. What many methods seek to do is avoid recomputing the entire condition of a rule by storing information on the partially computed condition of a rule between cycles.

In what follows, the example relations from figure 2 are used to illustrate two of the principal approaches. A typical production rule in Ariel notation [Hanson 1992] relating to this data could monitor every `UK` based owner of `IBM` stock:

```
if s.name = "IBM" and s.stock# = o.stock# and
   o.reg# = h.reg# and h.country = "UK"
   from s in Stock, o in Owns, h in Holder
do <action>
```

The query used to express the condition involves a join of the `Owns` relation with both `Stock` and `Holder`.

The principal notion behind the Rete [Forgy 1982] matching algorithm, which was originally proposed for implementing main-memory OPS-5 production rule systems, is that by storing the partially computed condition of a rule between cycles of rule execution, the effect of *changes* to the database on the conflict set can be computed with minimal additional effort. The partially computed condition of a rule is often stored in a graph structure known as a *discrimination network*. The example query above can be represented by the Rete discrimination network in figure 7.

The Rete network has a number of different node types – a *root* node serves as the entry point to the network; below the root node are a number of nodes that represent the base tables; below any node that represents a stored table, there can be zero or more filter nodes, each of which applies a single condition to the relation. After a filter has been applied, the tuples that have satisfied the condition in the filter are stored in α memory tables. A node with two parents performs a join on its parent relations, the result of which is stored in a β memory node. At the bottom of the tree is the result node that holds the conflict set.

The Rete match phase operates by passing information down through the network from the root. For example, if a new *Holder* tuple is entered into the database, it is tested to see if it has a *country* equal to *UK* – if so, it is stored in the α memory node *alpha1*. The tuple is then joined with the α memory node *alpha2*, and if tuples result from the join, they are stored in the β memory node *beta1*. Any such new

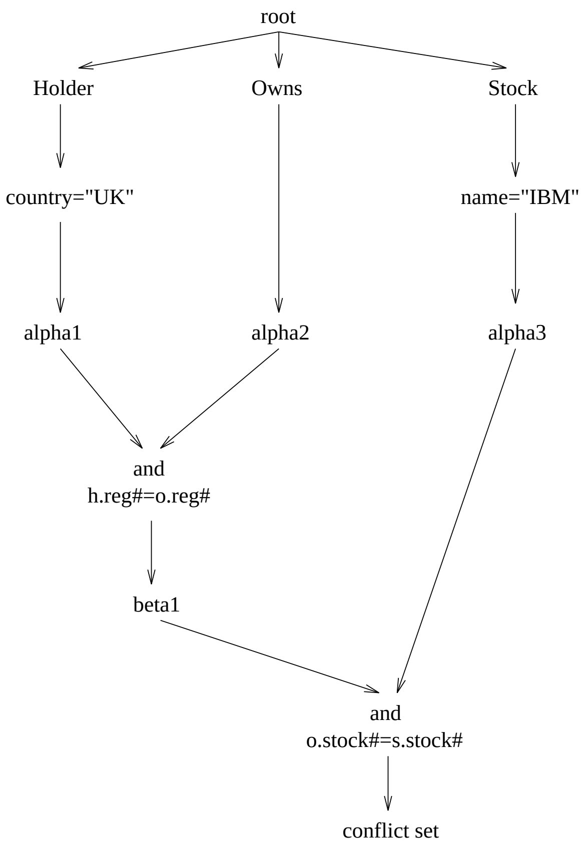

Fig. 7. Rete network for example rule

tuples are then joined to the α memory node *alpha3* to yield additional data for the conflict set.

While Rete has been used in commercial production-rule systems, there are a number of problems, especially when large databases are used: the space overhead is high, with both α and β memories storing the intermediate results of computation, and maintenance of such results, particularly on deletion of data, imposes a significant overhead. Work on the identification of alternatives to Rete has explored two principal issues relating to intermediate information:

What to store. A spectrum can be seen to exist, with no storage of intermediate results at one end, and storage of all intermediate results at the other. Rete is at the storage intensive end of this spectrum, with both α and β memories being used. A variation of Rete with α but no β memories is TREAT [Miranker 1987], which out-performs Rete in many cases [Wang and Hanson 1992]. More recent research has shown how rules can be analysed to identify what level of intermediate storage is most suitable for them [Fabret et al. 1993].

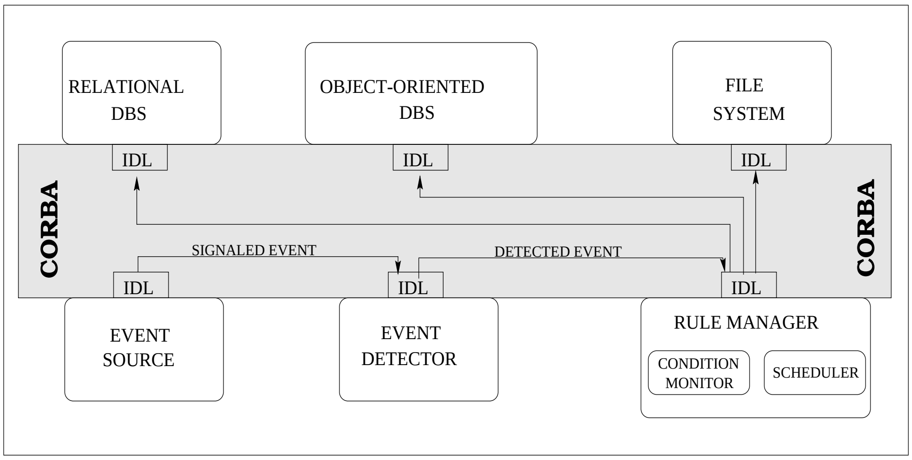

Fig. 8. Active rule system architecture: moving to a distributed setting.

How to store it. Algorithms such as TREAT can be built directly on top of relational storage structures, but researchers have identified structures that improve certain aspects of condition monitoring. For example, [Hanson 1992] exploits interval skip lists to reduce the cost of searching and updating α memories, and [Brant and Miranker 1993] uses a specialized index structure to improve join performance.

7.6 Active Rules in a Distributed Environment

Until now, the paper has assumed that active behavior is being supported in a single, centralized database. However, many modern applications have a distributed nature, and a number of proposals have been made for exploiting active behavior in distributed systems. Here, the active mechanism is seen as a set of cooperating objects distributed throughout a network of loosely-coupled autonomous nodes. Event detection, event management or rule management are seen as services that cooperate to offer the functionality previously bundled in a monolithic active mechanism. Standards such as CORBA [Orfali et al. 1996] provide the communication middleware required for cooperation in this distributed setting. Figure 8 gives an overview of how the architecture presented in figure 11 has been adapted to the new setting. The pioneer work of C^{2}offeine [Koschel et al. 1997] illustrates this approach. This system enables ECA rule based detection, processing and reporting of events and complex situations to be done in CORBA-based systems with heterogeneous and distributed information sources. Despite the limited experience to date, some insights can be given about the implications of distribution of active rule support.

As far as architecture is concerned, rule management is no longer centralized but distributed. Although figure 8 suggests than each component is centrally located at a different site, this does not need to be so. For instance, event detection can be distributed among sites [Bacon et al. 1995; Schwiderski 1996]. Each site contains a *local event detector* and a *global event detector*. The former basically corresponds to the detectors found in centralized systems, while the latter are responsible for monitoring inter-site composite events. A global event detector responsible for event e registers its interest in e's components in the corresponding detectors. The detection of a global event is then distributed among different sites. As soon as an event is detected, a signal is sent to each of the detectors that have registered an interest in it. Likewise, rule management support offers distinct alternatives: (1) a central rule manager maintaining a global rule base, (2) a set of rule managers in distinct sites each one with its local rule base, and (3) a set of rule managers but with a global rule set [von Bueltzingsloewen et al. 1998].

Being in a distributed context, architectural alternatives must consider communications overheads. For instance, CORBA includes an event service that supports asynchronous communication between objects. Communication is achieved through *event channels*. An event channel is a queue onto which events are placed by suppliers and removed by consumers. Two models are provided for event communication, based on who initiates the communication. The push model allows a supplier of events to initiate the transfer of the event data to consumers. The pull model permits a consumer of events to request the event data from a supplier[2]. These options should be considered when addressing communication among the distinct components supporting the active functionality. For instance, the signaling phase involving communication between event sources and the event detector, should it follow a push model (also known as re-active) or a pull model (also known as pro-active)? Re-active models are useful for central event detectors where every event is relevant for the system. Otherwise, an important communication overhead is caused since every event must be communicated to the event detector. On the other hand, pro-active models incur in a bigger processing overhead [von Bueltzingsloewen et al. 1998]. Likewise, communication between the event detector and the rule manager (the triggering phase) can also follow a re-active or a pro-active approach. The former implies explicit subscription of the rule manager to the set of events in which it is interested, whereas the pro-active approach leaves the rule manager to poll the event collection periodically [von Bueltzingsloewen et al. 1998].

Besides architectural considerations, both the knowledge and execution models should be also revised when moved to a distributed (and potentially, heterogeneous) setting. As for the knowledge model, heterogeneity of the event sources can lead to additional classifications besides those found in centralized systems (e.g. workflow events). Also, the diversity of the data sources and the lack of a single state, requires further precision in setting the context in which conditions and actions are performed. A complex problem is that of *time stamps*. In a centralized system there is a single local clock that allows a total ordering of event instances based on their occurrence time. That is, two distinct primitive events e_1 and e_2 can always be ordered as they cannot occur simultaneously: either e_1 happens before e_2 or vice versa. The time of a composite event occurrence is then derived from the occurrence times of its components. By contrast, distributed systems do not have global time:

[2]A key point, however, is that both the supplier and the consumer are aware of event processing i.e., they have to initiate event processing explicitly, thus violating the non-intrusive principle that characterizes the event-based paradigm. Moreover, CORBA does not explicitly support composite events.

each site has its own local clock, and events may occur simultaneously at different sites. As events contributing to a composite event originate from different sites, an event's timestamp should be globally meaningful. Therefore, local clocks must be *synchronized* through special time servers that transmit time information to each site, and the delay in the synchronization among the sites should be below a parameter P obtained as the maximum time difference between two "simultaneous" ticks of any two local clocks. Summing up, two time stamps are required, namely, *local* timestamps, needed for constructing intra-site composite events, and global timestamps as a canonical form used in the detection of inter-site composite events [Schwiderski 1996].

Finally, the execution model needs also to be revised. As a case in point, the *error handling* dimension should now be extended with communication delays and disruptions to be faced in any messaging among the components of the active mechanism. The solution will vary depending on the components and the impact of the error. For instance, communication delays during signaling between the event detector and an event source can cause global event detectors to receive events in a different order from their actual occurrence. Contingency actions can opt for *asynchronous evaluation* or *synchronous evaluation* [Schwiderski 1996]. The former ignores the fact that there may be delayed events and begins evaluation as soon as suitable event occurrences arrive at a composite event detector. Hence, events are not composed in a way that takes account of the order of their occurrence, but, on the other hand, event detection is not blocked by delayed events and is therefore faster. As an example, consider the event e_1 *or* e_2. If e_2's site fails, the disjunction can still be detected whenever e_1 is detected. By contrast, synchronous evaluation waits for delayed events, and evaluation proceeds only if *all* relevant events have arrived at the site of the global event detector. Although synchronous evaluation respects the occurrence order of events, the evaluation may be blocked long-term if there is transmission disruption or site failure. In the previous disjunctive example, the sites of both e_1 and e_2 are checked for relevant events before a disjunctive event occurrence is detected. In most cases, checking reveals that no relevant events should have been detected. Occasionally, a disruption can occur that delays the arrival of a relevant event. The global detector does this checking so that the consumption policy can be properly supported. As Schwiderski points out, which strategy is most appropriate depends on the application. Likewise, failure of data source access (e.g. due to a node breakdown) during condition evaluation or action execution should be addressed. The options proposed range from an *instead* action to the execution of no action [Koschel et al. 1997].

Although few details are yet available, distributed active rule management is likely to be one of the future services offered by CORBA-compliant suppliers. Its need can be corroborated by the proposals for event mechanisms which, independently from the active DBMS community, have already been proposed for supporting debugging [Bates 1995], communication and integration in distributed applications [Amouroux 1995; Barghouti and Krishnamurthy 1995].

8. DEVELOPING ACTIVE APPLICATIONS

Comprehensive support for active mechanisms within a database system is no guarantee that they will be used. Indeed, experience in the application of active databases often indicates that although such facilities are suitable for performing a range of different tasks, they are not straightforward to use. Difficulties include the following:

—It is not obvious (1) which parts of an application should be supported using active mechanisms, and which using other techniques, and (2) what performance penalty is likely to result from the use of rules. This is exacerbated by the lack of appropriate design methodologies.
—The functionality of a large rule base may be difficult to understand, with rules interacting in complex ways and no single description of how control flows through an application.
—The tools associated with an active rule system may be minimal, with little support for browsing, monitoring or debugging of active rules.

These points reflect a need for design methodologies, rule analysis techniques, and tools for debugging and explanation. In what follows, recent advances in each of these areas are outlined.

8.1 Rule Design

There are well-established techniques for database design that emphasize the description of the structural aspects of information in a domain (e.g. E/R modeling, normalization), and which in turn are used alongside mechanisms for describing processes (e.g. data flow diagrams) to capture the semantics of complete applications. Although active behavior is often closely linked to the structures stored in a database, and rule actions carry out tasks that can be viewed as processes, it is not yet clear what techniques are most suitable for, or most easily adapted to address, the functionalities supported by active rules. In particular, given the requirements of an application, design techniques should provide guidance on the aspects that should be supported using active mechanisms, and those that are better addressed using other facilities.

Where proposals have been made for methods with explicit support for active behavior, this has often been as an extension to an existing structural data model. For example, $(ER)^2$ [Navathe et al. 1995] is an extension of the ER model to support the description of events and rules that can be mapped to active rule facilities similar to those found in commercial relational databases. IFO_2 adapts the modeling constructs of the IFO semantic data model [Abiteboul and Hull 1987] for use describing composite events, and also includes a mapping to an active rule language, although in this case the target language must be more powerful than in the case of $(ER)^2$. More comprehensive facilities still, including temporal events, are provided by [Bichler and Schrefl 1994] in an extended object-oriented modeling language. These approaches, however, all assume that active rules should surface explicitly in the design method, which tends to prejudge the question of which parts of an application should be supported using active techniques and which using other alternatives.

This has been the focus of the IDEA methodology [Ceri and Fraternali 1997] which gives some initial insights on this topic, and provides high level description languages which in the later stages of development are mapped into low-level triggers (such as those found in a commercial DBMS).

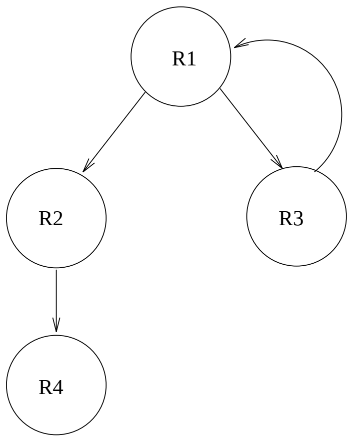

Fig. 9. Graph depicting possible rule triggering dependencies

8.2 Rule Analysis

The semantics of most active database systems in the literature are described informally – for representative examples see section 6, and papers such as [Stonebraker et al. 1990; Widom and Finkelstein 1990]. There are a number of disadvantages to this approach – specific descriptions may be incomplete or ambiguous, it is often difficult to compare the functionality of different systems, and there is no basis upon which to build tools that support the formal analysis of rule bases. Recently, formal specifications have been developed for active database systems using denotational semantics [Widom 1992; Coupaye and Collet 1995], Object-Z [Campin et al. 1997], and combined logic/operational techniques [Fraternali and Tanca 1995; Fernandes et al. 1997], but while such proposals help to clarify the informal definitions of specific systems, they have not yet been used as a basis for reasoning about rule bases.

There are a number of different characteristics of rule behavior for which a rule analyzer can search [Aiken et al. 1992]:

—**Termination:** Is rule processing guaranteed to terminate? Rule processing may fail to terminate whenever a cycle exists in a graph in which nodes represent rules and edges represent *Can-Trigger* relationships – the triggering graph. For example, in figure 9, any firing of rules *R2* and *R4* is guaranteed to terminate, but any firing of rules *R1* or *R3* may initiate a series of rule firings that fails to terminate.

Static analysis of a rule base can indicate whether a set of rules *may* fail to terminate. For example, a straightforward approach to the analysis of the *Can-Trigger* relationship would insert an edge from rule R_i to rule R_j if the action of R_i performs an update to a relation that is being monitored by the event of R_j. This approach is conservative, in the sense that potential non-termination will be detected even when in practice the rules will always terminate. For example, consider the following rule definition that relates to the unary relation `Rel` with the integer attribute `num`:

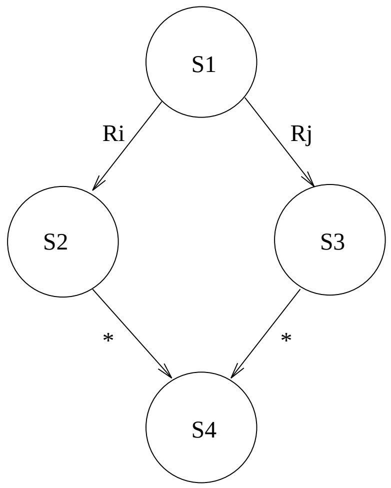

Fig. 10. Graph depicting confluent rule behavior.

```
create rule R1 as
on insert to Rel
if num > 10
do insert into Rel values (5)
```

Using the approach described above, `R1` in itself would be considered to be a potential source of non-termination, although in practice `R1` is never directly recursive because the tuple inserted by its action always leads to the condition of the rule being false. Less conservative approaches to rule termination analysis that examine the conditions and the actions of rules in more detail, but which are as a result more system specific, are presented in [Baralis and Widom 1994; Weik and Heurer 1995].

—**Confluence:** Is the result of rule processing independent of the order in which simultaneously triggered rules are selected for processing? If the policy `Holder` relation from section 2 had an attribute `NumStocks`, the role of this attribute could be interpreted differently by different rules. For example, the following rule would help to maintain `NumStocks` as the number of items of stock held:

```
on insert to Owns or update to reg# of Owns or
   update to reg# of Holder
if <change affects amount of stock held by Holder h>
do <update NumStocks attribute of h>
```

However, an alternative interpretation would treat `NumStocks` as the maximum number of stocks that a Holder can own at any one time, giving rise to the following rule:

```
on insert to Owns or update to reg# of Owns or
   update to reg# of Holder
if <amount of stock held by Holder h is more than NumStocks>
do <reduce range of stock held by Holder h>
```

These rules clearly implement inconsistent policies regarding stock holding; if both rules have the same priority and are present in the rule base at the same time then updates to the database will nondeterministically affect what stock is held.

Confluence can be understood by considering that the firing of a rule in one database state may lead to the creation of a new database state. If more than one rule is triggered at the same time, then more than one potential successor state exists, as illustrated in figure 10 where the states *S2* and *S3* are the successors to *S1* that result from the firing of rules R_i and R_j respectively. A rule base is confluent if for any two rules R_i and R_j triggered in any initial state *S1*, a single final state *S4* is guaranteed to be reached regardless of the order in which any subsequent simultaneously triggered rules are selected for firing (denoted as * in figure 10). Note that, for example, the actions of rules R_i and R_j may have triggered additional rules as a side-effect of the transition from *S1* to states *S2* and *S3*, and that the behaviors of these subsequently triggered rules have to be taken into account when considering confluence of the entire rule base.

Work on confluence analysis has developed algorithms for analyzing complete rule bases [Aiken et al. 1992], for considering the effect of an update on the truth of a condition [Baralis and Widom 1994; Levy and Sagiv 1993], and for characterizing the contexts in which non-confluent behavior may be exhibited [van der Voort and Siebes 1994].

—**Observable determinism:** Is the effect of rule processing as observed by a user of the system independent of the order in which triggered rules are selected for processing? This notion seeks to extend deterministic rule system behavior beyond the boundaries of the database itself. For example, the following two rules implement a response which is confluent but observably nondeterministic.

```
on <event E1>
if <condition C1>
do <send message to user>

on <event E1>
if <condition C1>
do <abort>
```

In this example, if the first rule is scheduled for firing before the second then the user receives a message and the transaction over the database is aborted. By contrast, if the second rule is scheduled before the first, then the transaction is aborted, but no message is sent to the user.

Initial research into the static analysis of active rule bases has not, to date, significantly eased the development of active applications. This is because such rule analyzers are not generally supplied with active systems, and because it is not always obvious what action should be taken when a potential source of non-termination or non-confluence is detected. There are a range of possible changes can be made – rule priorities can be used to tailor the order in which rules fire, rule conditions or actions can be modified to change their effect, or rules may be seen to be implementing conflicting policies and dropped. The development of effective rule analysis systems awaits further work on communicating the results of analysis to users, as well as experience based upon the use of implemented rule analyzers.

8.3 Rule Debugging

Users may be reluctant to apply active facilities because of anticipated problems with maintenance, unforeseeable behavior and lack of control [Simon and Kotz-Dittrich 1995], which motivates the development of debugging environments that help in defining safe rule sets i.e. rules that comply with the *termination, confluence* and *observable determinism* properties described above.

Unfortunately, these properties are not always easy to ascertain for a fully-fledged rule language at compile time, and hence research on rule analysis has focused on formal, declarative languages. However, not all authors agree that active rule systems should necessarily be associated with such languages, and many implemented rule systems are integrated with imperative database programming languages [Buchmann 1994]. Furthermore, the fact that a rule base exhibits terminating and confluent behavior does not in itself imply that it is correct. Thus, as rule languages become more complicated, thereby increasing the range of applications for which they are suitable, the need for rule debugging environments becomes increasingly pressing.

Unfortunately, traditional debugger models are not adequate for debugging rules. Conventional debuggers provide exhaustive information on the state of the execution process (e.g. program variables, subroutines and the like), thereby allowing the user to monitor the evolution of this state information. By contrast, what makes rule debugging a challenging task is the insidious ways in which rules can interact. Interaction, rather than state, becomes the main source of incorrect or unexpected behavior, and this *context-dependent* control exhibited by active rules imposes new demands on the debugger.

Unlike traditional programming languages, where sequential control is specified both explicitly and statically by the programmer, active rules are fired dynamically by the system based on the previous flow of events. There is thus no way to know in advance which rules will be fired. Rules eligible for firing, as represented by the *conflict set*, depend on the events raised (internal or external to the DBMS). Hence, it is more appropriate to reveal the context in which rules have been triggered (e.g. the conflict set and the event base) than to present a sequential trace of triggered rules. DEAR, a debugger for EXACT [Diaz et al. 1994], attempts to address this issue by showing the intertwined cycle of rules and events. Hence, the user can ascertain not only which rules have been triggered, but also whether the event triggering the rule was raised from a top-level transaction instruction or a rule execution. Furthermore, when an event is raised, such a cycle permits identification of the context in which the event took place in terms of recently triggered rules.

Figure 11 shows such an event-rule cycle for an immediate coupling mode. The representation is a tree where the root is artificially created (the corresponding node is labeled with *root*) and its direct descendents are the first events to be raised. Nodes can represent either events or rules, where event nodes alternate with rule nodes. An arc from an event node to a rule node means that the event has triggered the rule. An arc form a rule node to an event node means that the event was produced is a result of executing the action of the rule. As execu-

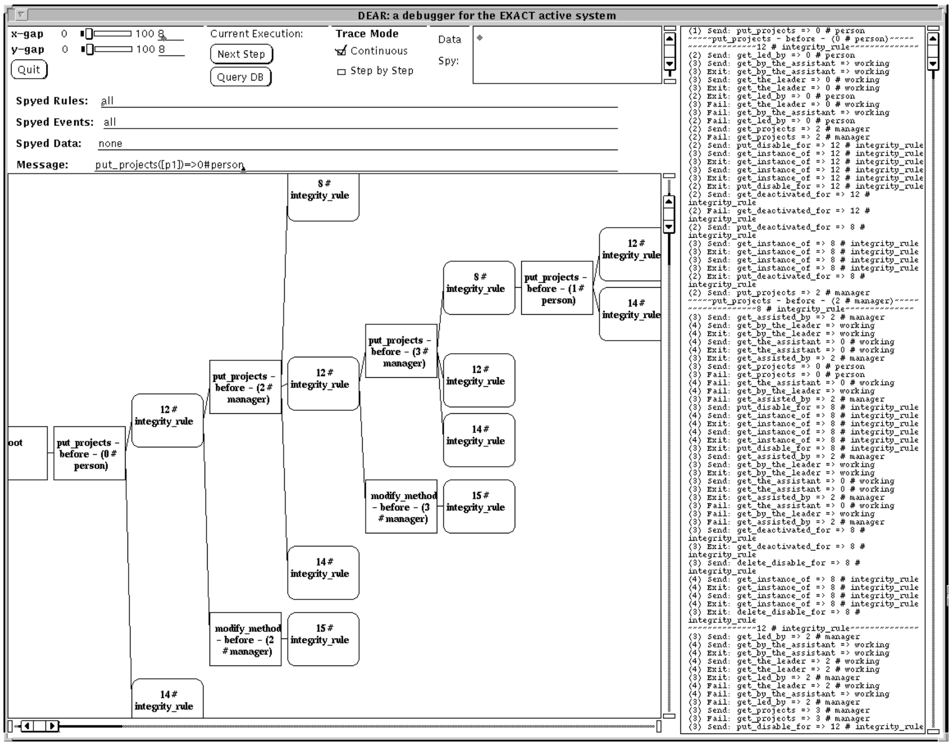

Fig. 11. An intertwined event-rule cycle.

tion proceeds, the event/rule tree is constructed depth-first. In figure 11, event *(put_projects,before,2#manager)* happened before event *(modify_method,before, 2# manager)*, and the leftmost occurrence of rule *12#integrity_rule* before the leftmost occurrence of rule *14#integrity_rule*. Moreover, figure 11 also makes explicit the conflict set of simultaneously triggered rules, once the conflicts among those rules have been resolved by the rule manager. An indication of the rules participating in this conflict set is useful for focusing on a restricted number of rules where complex interactions can occur.

Rather than building trees following the execution, an alternative investigated in the visual tool for rule analysis VITAL [Benazet et al. 1995], is to show the trace in the triggering graph created during analysis (see figure 9). A color code is used for each event state (i.e. raised or not raised) and each rule state (i.e. inactive, triggered or executed). As execution proceeds the colors displayed change, and a textual trace of the execution is displayed in a separated window. VITAL also proposes the use of a *statistics manager* that calculates and records statistics on the behavior of the rule processor (e.g. the number of tuples inserted, deleted or modified during a rule execution cycle, the number of times a rule is triggered, etc). This data can be used, for instance, to identify potentially infinite cycles, or to help find erroneous rule declarations. The former can be ascertained by monitoring the size of relevant tables: if the size tends to increase in a uniform way, it suggests that the cycle will not terminate. Erroneous rule condition declarations may be suspected if the number of times a rule is executed is very low compared with the number of times the rule is triggered. Thus, features of statistical data can point to potentially abnormal behavior.

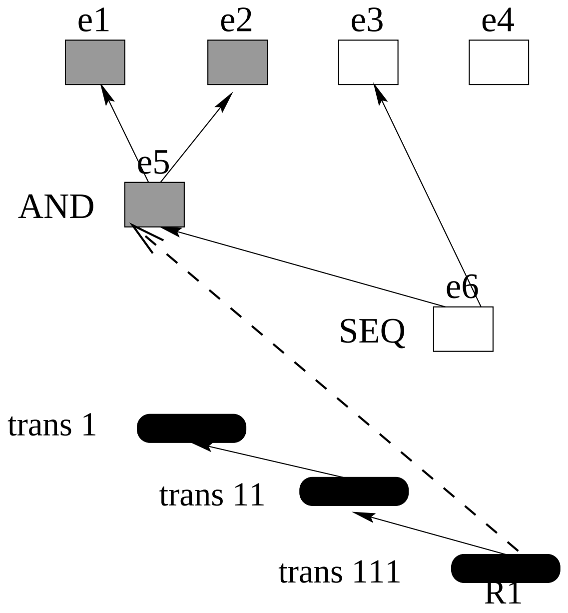

Fig. 12. Tracing composite events in Sentinel.

The above approaches have mainly focused on displaying the interleaving of events and rules, but are restricted to primitive events. The monitoring of composite event occurrences is a more complicated task due to the large number of occurrences that may need to be shown and the intricate ways in which events are combined to obtain composite event occurrences. A first approach to this issue has been presented as part of the Sentinel system [Chakravarthy et al. 1995]. Sentinel provides a post-execution debugging tool where information from the execution is stored in log files that are consulted by the debugging tool to simulate runtime activities. An event tree is created where primitive events are leaf nodes and composite events are seen as parents of their component events. A color code is used to represent event status (detected or not detected). This tree grows from primitive events to the root. In addition, a transaction tree describes the triggering rule context: the root node represents the top-level transaction, and child nodes represent rules fired in the context of their parent node (either the user transaction or a rule, as rules are executed as subtransactions). A color code is used to represent the different states of subtransactions: running, suspended or committed. Whenever a rule is fired, a line is drawn connecting the transaction node of the current rule and the triggering event. An example is given in figure 12. The detection of events *e1* and *e2* leads to the happening of the conjunction event *e5*, which in turn causes the triggering of rule *R1* (i.e. *trans 111*). This rule is fired in the context of the running transaction, *trans 11*, which is a child of transaction *trans 1*. In the first version of this system, all rules and events produced are shown, which could lead to cluttered visualizations [Chakravarthy et al. 1995].

Despite the advances in rule debugging, more remains to be done (e.g. customizable visualization of rule execution, automatic generation of test data), and different active rule system functionalities are likely to be most effectively presented to users using different visualizations.

9. CONCLUSIONS

Research into active databases has proceeded along similar lines to early research into object-oriented databases, in that there has been considerable experimentation, but relatively little work on standardization or theory. This has resulted in a wide range of constructs, execution strategies and software architectures being proposed that have utility in different problem domains [Dittrich et al. 1995]. This paper has reviewed research and development work in active databases by: describing tasks that can benefit from active behavior, presenting a framework that characterizes important aspects of active functionality, describing a range of representative systems within the framework, indicating how implemented systems support the principal features described in the framework, and by outlining ongoing activity on tools that support the design and implementation of applications that exploit active technologies. In so doing, the aim has been to identify the principal contributions made by researchers to date, to indicate how important ideas have made their way into implemented systems, to allow detailed comparison of specific proposals, and to suggest areas that stand to benefit from future research results.

Future research is required on: application and efficient implementation of event algebras, optimization of active rules, implementation and use of rule analysers, tools that support design and maintenance of rule bases, architectures for real-time active applications, distributed active functionality, and integration of active behavior with deductive and temporal facilities.

Acknowledgements We are pleased to acknowledge the support of the European Union Human Capital and Mobility Network ACT-NET, the UK Engineering and Physical Sciences Research Council (grant GR/H43847) and the Basque Government for funding active database research involving the authors. We are also grateful to our colleagues for useful discussions on active database systems, including Alex Buchmann, Andrew Dinn, Alvaro Fernandes, Ray Fern'andez, Peter Gray, Jon Iturrioz, Arturo Jaime and Howard Williams.

REFERENCES

Abiteboul, S. and Hull, R. 1987. IFO: A Formal Semantic Database Model. *ACM Transactions on Database Systems 12*, 4 (December), 525–565.

Agrawal, R., Cochrane, R., and Lindsay, B. 1991. On maintaining priorities in a production rule language. In G. Lohman, A. Sernadas, and R. Camps Eds., *Proc. 17th VLDB* (1991), pp. 479–487. Morgan-Kaufmann.

Aiken, A., Widom, J., and Hellerstein, J. 1992. Behaviour of database production rules: Termination, confluence, and observable determinism. In *ACM SIGMOD*, Volume 21 (1992), pp. 59–68.

Amouroux, R. 1995. Reactive services for supporting tool integration in a development environment. In *Proc on Technology of Object-Oriented Languages and Systems (TOOLS)* (1995), pp. 61–70.

BACON, J., BATES, J., HAYTON, R., AND MOODY, K. 1995. Using events to build distributed applications. In *Proc. Int. Workshop on Services in Distributed and Networked Environments (SDNE), Whistler, British Columbia* (1995), pp. 148–155.

BARALIS, E. AND WIDOM, J. 1994. An algebraic approach to rule analysis in expert database systems. In J. BOCCA, M. JARKE, AND C. ZANIOLO Eds., *Proc. 20th VLDB* (1994), pp. 475–486. Morgan-Kaufmann.

BARALIS, E. AND WIDOM, J. 1995. Using Delta Relations to Optimize Condition Evaluation in Active Databases. In T. SELLIS Ed., *Proc. 2nd Int. Wshp. on Rules In Database Systems (RIDS)* (1995), pp. 292–308. Springer-Verlag.

BARGHOUTI, N. AND KRISHNAMURTHY, B. 1995. Using event contexts and matching constraints to monitor software processes. In *Proc. ICSE* (1995), pp. 83–92.

BATES, P. 1995. Debugging heterogeneous distributed systems using event-based models of behaviour. *ACM Transactions on Computer Systems 13*, 1, 1–31.

BAYER, P. AND JONKER, W. 1994. A framework for supporting triggers in deductive databases. In N. PATON AND M. WILLIAMS Eds., *Proc. 1st Int. Workshop on Rules In Database Systems* (1994), pp. 316–330. Springer-Verlag.

BEERI, C. AND MILO, T. 1991. A model for active object oriented database. In R. C. G.M. LOHMAN, A. SERNADAS Ed., *17th Intl. Conf. on Very Large Data Bases, Barcelona* (1991), pp. 337–350. Morgan Kaufmann (ISBN 1-55860-150-3).

BENAZET, E., GUEHL, H., AND BOUZEGHOUB, M. 1995. VISUAL: a visual tool for analysis of rule behaviour in active databases. In T. SELLIS Ed., *Proc. 2nd Int. Wshp. On Rules in Database Systems* (1995), pp. 182–196. Springer-Verlag.

BICHLER, P. AND SCHREFL, M. 1994. Active object-oriented database using active object/behaviour diagrams. In *Proceedings of the Fourth International Workshop on Research Issues in Data Engineering (RIDE-ADS'94)* (1994), pp. 163–171.

BLUE, A., BROWN, B., AND GRAY, W. 1988. An implementation of alerters for health district management. In W. GRAY Ed., *Proc. 6th British National Conference on Databases (BNCOD)* (1988), pp. 125–140. CUP.

BRANDING, H., BUCHMANN, A., KUDRASS, T., AND ZIMMERMANN, J. 1994. Rules in an Open System: The REACH Rule System. In N. PATON AND M. WILLIAMS Eds., *Rules in Database Systems* (1994), pp. 111–126. Springer-Verlag.

BRANT, D. AND MIRANKER, D. 1993. Index support for rule activation. In *SIGMOD* (1993), pp. 42–48. ACM.

BUCHMANN, A. 1994. Current trends in active databases: are we solving the right problems. In C. CHRISMENT Ed., *Information Systems Design and Multimedia (Proc. Basque International Workshop on IT)* (1994), pp. 121–133. Cepadues Editions.

BUCHMANN, A., ZIMMERMANN, J., BLAKELY, J., AND WELLS, D. 1995. Building an integrated active oodbms: Requirements, architecture, and design decisions. In *Proc. IEEE Data Engineering* (1995), pp. 117–128.

CAMPIN, J., PATON, N., AND WILLIAMS, M. 1997. Specifying Active Database Systems in an Object-Oriented Framework. *Software Engineering and Knowledge Engineering 7(1)*, 101–123.

CERI, S. AND FRATERNALI, P. 1997. *Designing Applications with Objects and Rules: the IDEA Methodology.* International Series on Database Systems and Applications, Addison-Wesley Longman.

CERI, S., FRATERNALI, P., PARABOSCHI, S., AND TANCA, L. 1996. Active Rule Management in Chimera. In J. WIDOM AND S. CERI Eds., *Active Database Systems: Triggers and Rules for Active Database Processing* (1996), pp. 151–175. Morgan Kaufmann.

CERI, S., GOTTLOB, G., AND TANCA, L. 1990. *Logic Programming and Databases.* Springer-Verlag, Berlin.

CERI, S. AND WIDOM, J. 1991. Deriving production rules for incremental view maintenance. In R. C. G.M. LOHMAN, A. SERNADAS Ed., *17th Intl. Conf on Very Large Data Bases* (1991), pp. 577–589. Morgan Kaufmann.

CERI, S. AND WIDOM, J. 1993. Managing semantic heterogeneity with production rules and persistent queries. In R. AGRAWAL, S. BAKER, AND D. BELL Eds., *19th Intl. Conf on Very Large Data Bases* (1993), pp. 108–119. Morgan Kaufmann.

CHAKRAVARTHY, S. 1989. Rule management and evaluation: an active *DBMS* perspective. *SIGMOD RECORD 18*, 3, 20–28.

CHAKRAVARTHY, S., ANWAR, E., MAUGIS, L., AND MISHRA, D. 1994. Design of Sentinel: an object-oriented DBMS with event-based rules. *Information and Software Technology 36*, 9, 555–568.

CHAKRAVARTHY, S., KRISHNAPRASAD, V., ANWAR, E., AND KIM, S.-K. 1994. Composite events for active databases: Semantics, contexts and detection. In J. BOCCA, M. JARKE, AND C. ZANIOLO Eds., *Proc. 20th Int. Conf. on Very Large Data Bases* (1994), pp. 606–617. Morgan-Kaufmann.

CHAKRAVARTHY, S., TAMIZUDDIN, Z., AND ZHOU, J. 1995. A visualization and explanation tool for debugging ECA rules in active databases. In T. SELLIS Ed., *Proc. 2nd Int. Wshp. On Rules in Database Systems* (1995), pp. 196–209. Springer-Verlag.

CHANDRA, R. AND SEGEV, A. 1994. Active Databases for Financial Applications. In J. WIDOM AND S. CHARAVARTHY Eds., *Proc. 4th Int. Workshop on Research In Data Engineering (RIDE-ADS)* (1994), pp. 46–52. IEEE.

COLLET, C., COUPAYE, T., AND SVENSEN, T. 1994. NAOS: Efficient and modular reactive capabilities in an object-oriented database system. In J. BOCCA, M. JARKE, AND C. ZANIOLO Eds., *Proc. 20th VLDB Conf.* (1994), pp. 132–143. Morgan-Kaufmann.

COLLET, C. AND MANCHADO, J. 1995. Optimization of Active Rules With Parallelism. In M. BERNDTSSON AND J. HANSSON Eds., *Proc. Active and Real Time Database Systems (ARTDB)* (1995), pp. 82–103. Springer-Verlag.

COUPAYE, T. AND COLLET, C. 1995. Denotational Semantics for and Active Rule Execution Model. In T. SELLIS Ed., *Proc. 2nd Int. Workshop on Rules in Database Systems* (1995), pp. 36–50. Springer-Verlag.

DAYAL, U. 1989. Active database management systems. *SIGMOD RECORD 18*, 3, 150–169.

DAYAL, U., BUCHMANN, A., AND MCCARTHY, D. 1988. Rules are objects too: A knowledge model for an active object oriented database system. In K. DITTRICH Ed., *Proc. 2nd Intl. Workshop on OODBS*, Volume 334 (1988), pp. 129–143. Springer-Verlag. Lecture Notes in Computer Science.

DAYAL, U., HSU, M., AND LANDIN, R. 1990. Organising Long-Running Activities with Triggers and Transactions. In *SIGMOD Conference* (1990), pp. 204–214. ACM.

DEUX, O. AND ET AL. 1990. The Story of O_2. *IEEE Transactions on Knowledge and Data Engineering 2*, 1 (March), 91–108.

DIAZ, O. 1992. Deriving rules for constraint maintenance in an object-oriented database. In I. R. A.M. TJOA Ed., *Proc. Intl. Conf. on Databases and Expert Systems DEXA* (1992), pp. 332–337. Springer-Verlag.

DIAZ, O. AND JAIME, A. 1997. EXACT: an EXtensible approach to ACTive object-oriented databases. *VLDB Journal 6*, 4, 282–295.

DIAZ, O., JAIME, A., AND PATON, N. 1994. DEAR: A DEbugger for Active Rules in An Object-Oriented Context. In N. PATON AND M. WILLIAMS Eds., *Proc. 1st Int. Wshp on Rules In Database Systems* (1994), pp. 180–193. Springer-Verlag.

DIAZ, O., JAIME, A., PATON, N., AND AL QAIMARI, G. 1994. Supporting Dynamic Displays Using Active Rules. *SIGMOD Record 23*, 1, 21–26.

DIAZ, O., PATON, N., AND GRAY, P. 1991. Rule management in object oriented databases: a uniform approach. In G. LOHMAN, A. SERNADAS, AND R. CAMPS Eds., *17th Intl. Conf. on Very Large Data Bases, Barcelona* (1991), pp. 317–326. Morgan Kaufmann.

DINN, A., PATON, N., WILLIAMS, M., AND FERNANDES, A. 1996. An Active Rule Language for ROCK & ROLL. In *Proc. 14th British National Conference on Databases* (1996), pp. 36–55. Springer-Verlag.

DITTRICH, A. K. 1993. Adding Active Functionality to an Object-Oriented Database – a Layered Approach. In *Proc. Datenbanksysteme in Buro, Braunschweig, Germany* (1993).

DITTRICH, K., GATZIU, S., AND GEPPERT, A. 1995. The Active Database Management System Manifesto: A Rulebase of ADBMS Features. In T. SELLIS Ed., *Rules In Database Systems: Proc. of the 2nd Int. Workshop* (1995), pp. 3–17. Springer-Verlag.

ETZION, O. 1993. *PARDES* – a data-driven oriented active database model. *SIGMOD RECORD 22*, 1, 7–14.

ETZION, O., GAL, A., AND SEGEV, A. 1994. Data Driven and Temporal Rules in PARDES. In N. PATON AND M. WILLIAMS Eds., *Proc. 1st Int. Workshop on Rules In Database Systems* (1994), pp. 92–108. Springer-Verlag.

FABRET, F., REGNIER, M., AND SIMON, E. 1993. An Adaptive Algorithm for Incremental Evaluation of Production Rules in Databases. In R. AGRAWAL, S. BAKER, AND D. BELL Eds., *19th Intl. Conf. on Very Large Data Bases* (1993), pp. 455–466. Morgan Kaufmann.

FERNANDES, A., WILLIAMS, M., AND PATON, N. 1997. A Logic-Based Integration of Active and Deductive Databases. *New Generation Computing 15*, 2, 205–244.

FERNANDEZ, R. AND DIAZ, O. 1995. Reactive behaviour support: Themes and variations. In T. SELLIS Ed., *Proc. 2nd Int. Wshp. On Rules in Database Systems* (1995), pp. 69–85. Springer-Verlag.

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

FRATERNALI, P. AND TANCA, L. 1995. A structured approach to the definition of the semantics of active databases. *ACM TODS 20*, 4, 414–471.

GATZIU, S. AND DITTRICH, K. 1994. Events in an active object-oriented database. In N. PATON AND M. WILLIAMS Eds., *Proc. 1st Int. Workshop on Rules in Database Systems* (1994), pp. 23–39. Springer-Verlag.

GATZIU, S., GEPPERT, A., AND DITTRICH, K. 1991. Integrating active concepts into an object-oriented database system. In P. KANELLAKIS AND J. SCHMIDT Eds., *Proc. 3^{rd} Workshop on Database Programming Languages* (1991). Morgan-Kaufmann.

GEHANI, N. AND JAGADISH, H. 1991. ODE as an Active Database: Constraints and Triggers. In R. C. G.M. LOHMAN, A. SERNADAS Ed., *17th Intl. Conf. on Very Large Data Bases, Barcelona* (1991), pp. 327–336. Morgan Kaufmann.

GEHANI, N., JAGADISH, H., AND SHMUELI, O. 1992. Event specification in an active object-oriented database. *ACM SIGMOD*, 81–90.

GEPPERT, A., BERNDTSSON, M., LIEUWEN, D., AND RONCANCIO, C. 1996. Performance evaluation of object-oriented active database management systems using the beast benchmark. Technical Report 96.07 (October), Department of Computer Science, University of Zurich.

GEPPERT, A. AND DITTRICH, K. 1994. Rule-based implementation of transaction model specifications. In N. PATON AND M. WILLIAMS Eds., *Proc. 1st Int. Workshop on Rules In Database Systems* (1994), pp. 127–142. Springer-Verlag.

HANSON, E. 1992. Rule Condition Testing and Action Execution in Ariel. In *Proc. SIGMOD* (1992), pp. 49–58. ACM.

HANSON, E. N. AND WIDOM, J. 1993. An overview of production rules in database systems. *The Knowledge Engineering Review 8*, 2, 121–143.

HARDER, T. AND ROTHERMEL, K. 1993. Concurrency control issues in nested transactions. *VLDB Journal 2*, 1, 39–74.

HARRISON, J. AND DIETRICH, S. 1994. Integrating active and deductive rules. In N. PATON AND M. WILLIAMS Eds., *Proc. 1st Int. Workshop on Rules In Database Systems* (1994), pp. 288–305. Springer-Verlag.

HSU, M., LADIN, R., AND MCCARTHY, D. 1988. An execution model for active data base management systems. In *Proc. Int. Conf. on Data and Knowledge Bases* (1988), pp. 171–179.

KIERNAN, G., DE MAINDREVILLE, C., AND SIMON, E. 1990. Making Deductive Databases a Practical Technology: a step forward. In H. GARCIA-MOLINA AND H. JAGADISH Eds., *Proc.*

ACM SIGMOD Conf (1990), pp. 237–246.

KIM, W., LEE, Y., AND SEO, J. 1992. A Framework For Supporting Triggers in Object-Oriented Database Systems. *Int. Journal of Intelligent and Cooperative Information Systems 1*, 1, 127–143.

KOSCHEL, A., KRAMER, R., VON BULTZINGSLOEWEN, G., BLEIBEL, T., KRUMLINDE, P., SCHMUCK, S., AND WEINAND, C. 1997. Configurable active functionalty for *corba*. In *11th ECOOP Workshop 7 on CORBA* (1997).

KOTZ, A., DITTRICH, K., AND MULLE, J. 1988. Supporting semantic rules by a generalized event/trigger mechanism. In *Advance in Database Technology, EDBT, Venice* (1988), pp. 76–91.

KULKARNI, K., MATTOS, N., AND COCHRANE, R. 1998. Active Database Features in SQL-3. In N. PATON Ed., *Active Rules in Databases* (1998). Springer-Verlag.

LEVY, A. AND SAGIV, Y. 1993. Queries independent of updates. In R. AGRAWAL, S. BAKER, AND D. BELL Eds., *Proc. 19th VLDB* (1993), pp. 171–181. Morgan-Kaufmann.

MIRANKER, D. 1987. TREAT: A Better Match Algorithm for AI Production Systems. In *Proc. AAAI* (1987), pp. 42–47.

MOSS, E. Ed. 1985. *Nested Transactions: An Approach to Reliable Distributed Computing.* MIT Press.

NAQVI, W. AND IBRAHAM, M. 1994. Rule and knowledge management in an active database system. In N. PATON AND M. WILLIAMS Eds., *Proc. 1st Int. Workshop on Rules In Database Systems* (1994), pp. 58–69. Springer-Verlag.

NAVATHE, S., TANAKA, A., MADHAVAN, R., AND GAN, Y. H. 1995. A methodology for application design using active database technology. In *Report RL-TR-95-41 from the Rome Laboratory* (1995).

ORFALI, R., HARKEY, D., AND EDWARDS, J. 1996. *The Essential Distributed Objects Survival Guide.* John Wiley & Sons, Inc.

PATON, N. Ed. 1998. *Active Rules In Databases.* Springer-Verlag.

PATON, N., DIAZ, O., AND BARJA, M. 1993. Combining active rules and metaclasses for enhanced extensibility in object-oriented systems. *Data and Knowledge Engineering 10*, 45–63.

PATON, N., DIAZ, O., WILLIAMS, M., CAMPIN, J., DINN, A., AND JAIME, A. 1994. Dimensions of active behaviour. In N. PATON AND M. WILLIAMS Eds., *Proc. 1st Int. Workshop on Rules In Database Systems* (1994), pp. 40–57. Springer-Verlag.

PATON, N., DOAN, D., DIAZ, O., AND JAIME, A. 1996. Exploitation of object-oriented and active constructs in database interface development. In J. KENNEDY AND P. BARCLAY Eds., *Proc. 3nd Int. Workshop on Interfaces to Database Systems* (1996). Springer-Verlag.

REDDI, S., POULOVASSILIS, A., AND SMALL, C. 1995. Extending a Functional DBPL With ECA-Rules. In T. SELLIS Ed., *Proc. 2nd Int. Wshp. on Rules in Database Systems* (1995), pp. 101–115. Springer-Verlag.

SCHWIDERSKI, S. 1996. *Monitoring the Behaviour of Distributed Systems.* PhD thesis, University of Cambridge, United Kingdom.

SIMON, E. AND KOTZ-DITTRICH, A. 1995. Promises and realities of active database systems. In U. DAYAL, P. GRAY, AND S. NISHIO Eds., *Proc. 21st Int. Conf. on Very Large Data Bases* (1995), pp. 642–653. Morgan-Kaufmann.

SKOLD, M. AND RISCH, T. 1995. Using Partial Differencing for Efficient Monitoring of Deferred Complex Rule Conditions. In *Proc. IEEE Data Engineering* (1995).

STONEBRAKER, M., JHINGRAN, A., GOH, J., AND POTAMIANOS, S. 1990. On rules, procedures, caching and views in database systems. In *Proc. ACM SIGMOD* (1990), pp. 281–290.

STONEBRAKER, M. AND KEMNITZ, G. 1991. The POSTGRES Next-generation Database Management System. *Communications of the ACM 34*, 10 (October), 78–92.

THOMAS, I. AND JONES, A. 1995. Design and implementation of and active object-oriented database supporting construction of database tools. In T. SELLIS Ed., *Proc. 2nd. Int. Wshp. on Rules In Database Systems* (1995), pp. 147–164. Springer-Verlag.

van der Voort, L. and Siebes, A. 1994. Enforcing confluence of rule execution. In N. Paton and M. Williams Eds., *Proc. 1st Int. Workshop on Rules In Database Systems* (1994), pp. 194–207. Springer-Verlag.

von Bueltzingsloewen, G., Koschel, A., Lockemann, P., and Walter, H. 1998. *eca* functionality in a distributed environment. In *in [Paton 1998]* (1998).

Wang, Y.-W. and Hanson, E. 1992. A Performance Comparison of the Rete and TREAT Algorithms for Testing Database Rule Conditions. In *Proc. Data Engineering* (1992), pp. 88–97. IEEE.

Weik, T. and Heurer, A. 1995. An Algorithm for the Analysis of Termination of Large Trigger Sets in an OODBMS. In M. Berndtsson and J. Hansson Eds., *Proc. Active Real Time Database Systems (ARTDB)* (1995), pp. 158–169. Springer-Verlag.

Wells, D., Blakeley, J., and Thompson, C. 1992. Architecture of an Open Object-Oriented Database Management System. *IEEE Computer 25*, 10 (October).

Widom, J. 1992. A Denotational Senmantics for the Starburst Production Rule Language. *ACM SIGMOD Record 21*, 3, 4–9.

Widom, J., Cochrane, R., and Lindsay, B. 1991. Implementing Set-Oriented Production Rules as an Extension to Starburst. In R. C. G.M. Lohman, A. Sernadas Ed., *17th Intl. Conf. on Very Large Data Bases, Barcelona* (1991), pp. 275–286. Morgan Kaufmann (ISBN 1-55860-150-3).

Widom, J. and Finkelstein, S. 1990. Set-Oriented Production Rules in Relational Database Systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (1990), pp. 259–270.

Winston, P. 1984. *Artificial Intelligence (Second Edition)*. Addison Wesley.

Zaniolo, C. 1994. A unified semantics for active and deductive databases. In N. Paton and M. Williams Eds., *Rules in Database Systems* (1994). Springer-Verlag.