The Knowledge Engineering Review, Vol 8:2, 1993, 145–171 © Cambridge University Press

State-of-the-art report on reactive processing in databases and artificial intelligence

PETRA BAYER

ECRC, Arabellastr. 17, D-8000 Munich 81, Germany

Abstract

Active rules have been a standard technique in artificial intelligence (AI) for almost two decades. Variants of the AI methods are currently being adapted to provide database systems with the ability to respond reactively to events and database state changes. This paper gives an overview of developments in reactive processing database research, concentrating on active databases that are integrated with relational and object-oriented systems. A general presentation of "trigger"-based processing techniques is given, with a detailed review of active relational and object-oriented database models. An overview of technologies for active processing in AI is also presented, and some common and contrasting themes in database and AI technology identified.

Introduction

Active rules have been a standard technique in artificial intelligence (AI) for almost two decades. Variants of the AI methods are currently being adapted to provide database systems with the ability to respond to events and database state changes. This report presents an overview of reactive processing in the database field, concentrating on active databases integrated with relational and object-oriented database systems. A historical review of AI methods is also presented, to highlight similarities and differences with the database techniques.*

Part 1 of this report describes reactive processing in database research. Following an introduction (section 1), a general description of trigger languages and trigger processing is given in section 2, summarizing the main approaches to triggers that are being investigated. Different methods of conflict resolution in trigger processing and transaction handling are presented, and problems for termination and determinism of trigger processing identified. Section 3 gives descriptions of active database systems based on relational database models (Postgres: Stonebraker et al., 1989; Starburst: Widom et al., 1991; Ariel: Hanson, 1992; RDL1: Kiernan et al., 1990; Alert: Schreier, et al., 1991); section 4 presents active database systems based on object-oriented database models (HiPAC: McCarthy & Dayal, 1989; Ode: Gehani & Jagadish, 1991; Tanguy: Czejdo et al., 1991). Part 2 of the report gives a short overview of reactive processing in AI. Section 5 identifies the historical roots of the method, notably in Newell and Simon's work in cognitive modelling (section 6). Technologies for active processing in applied AI are presented in section 7. Some common themes are identified in section 8. Finally we discuss some points of contrast between the technologies developed in the two fields.

Part I: Reactive processing in database systems

1 Introduction

Database systems (DBS) support the storage, update and retrieval of large amounts of data. In general, DBSs can be distinguished according to whether they are based on relational, object-oriented or deductive models. Relational database systems are well-known and many commercial

*This section was drafted by John Fox of the Imperial Cancer Research Fund, London.

systems exist (e.g., Ingres, Held 1975). Object-oriented (e.g.O_2, Bancilhon et al., 1992) and deductive database systems (e.g., Aditi, Vaghani et al., 1990; EKS-V1, Vielle et al., 1991) emerged because of deficiencies of relational database systems. Object-oriented database systems exploit the idea of encapsulation of data and operations on the data, using the object concept developed in object-oriented programming languages. In deductive database systems, knowledge can be represented either explicitly by means of facts, or implicitly by means of derivation rules (Ceri et al., 1990). Furthermore integrity constraints can be used to specify semantically correct database states. These approaches for database systems mainly *aim to increase modelling facilities and expressiveness* of database systems instead of forcing the user to program the desired features in application programs.

Conventional database systems are generally passive, with queries and updates only executed when the user explicitly requires them. Designers of active database systems (ADBS) are now trying to *transform database systems from passive data storage facilities into active systems that can react immediately to events.* ADBSs can be characterized by the ability to monitor database states, and to react according to pre-defined actions, e.g., data manipulation operations, or procedure calls. These are executed automatically when a specified event occurs (e.g., another database operation) and a certain condition of the database becomes satisfied. Neither monitoring of the pre-defined conditions nor initiation of the various actions resulting from the satisfaction of some conditions should require user interaction. This means that, in addition to standard database facilities like transaction handling, concurrency, data manipulation, and querying, a new facility is offered and has to be integrated.

Constructs to specify active behaviour are called alerters (Buneman & Clemons, 1979), triggers (Beeri et al., 1991), demons (Caseau, 1991), event-condition-action rules (Rosenthal et al., 1989), or production rules (Stonebraker et al., 1989).* Triggers can increase the functionality of a database system and simplify the modelling of applications, when specifying events and conditions which require (re)actions to occur automatically. Designers of ADBS claim that triggers provide a uniform mechanism to enforce integrity constraints, control access rights, maintain versions of data, and so on, and that they are basic requisites for building large knowledge-based systems, though it has also been claimed (Zhou & Hsu, 1990; Aiken et al., 1992), that even with a small set of triggers it becomes difficult to follow the behaviour of the triggers.

In the presentation of reactive processing in databases we concentrate on ADBSs integrated with relational or object-oriented database systems. We do not consider deductive database systems, though some work has also been done on extending deductive databases with triggers. Caseau (1991) describes how to combine triggers with a deductive object-oriented database, and illustrates the advantages of a combination of procedural attachments and logic programming. Cacace and Ceri (1990) describe how "active constraints" can be integrated into a database system that integrates the object-oriented data modelling paradigm with a rule based paradigm based on an extension of Datalog (Ceri et al., 1989). The RDL1 system can also be regarded as a deductive database system where triggers are used to implement a deductive rule language. Ceri and Widom (1992a) show how deductive rules can be implemented by triggers, relying on the facilities of an ADBS.

This report does not address the problem of trigger handling in parallel or distributed database environments; these are discussed, for example, in Ceri and Widom (1992b) or Bahr et al. (1992).

2 Trigger languages and processing

2.1 Trigger syntax

Typically, triggers in database systems have the syntax

"ON event IF condition THEN action"

*Following common usage we refer to these constructs as *triggers* in the discussion of databases, and productions or production rules in the section on AI.

If the event occurs, the condition is checked in the current database state, and if the condition is satisfied, the action is executed.

An *event* is a happening of interest which is supposed to initiate trigger processing. In the relational database context events are usually generic data modifications like insertions or deletions of tuples. In the object-oriented database context, the creation or deletion of objects, or method invocations on objects are regarded as events. In addition, some systems allow clock ticks or external notifications by application programs to serve as events. When classifying events one can distinguish them according to temporal events (e.g., at 9 o'clock), external events (e.g., a temperature value), events related to transactions (e.g., the start of a transaction), to database manipulation operation (e.g., the modification of data), to data retrieval operations (e.g., the retrieval of some data), to operations on objects or classes (e.g., the creation of an object), and so on. These basic events can also be combined to build composite events. A detailed specification of an event language for the composition of events can be found in Gehani et al. (1992, 1992a). They propose to use finite automata to efficiently detect composite events.

A *condition* is a database query which is evaluated over the database state, when an event occurs. If there are one or more answers for the query, then the condition is said to be satisfied. If variables of the condition occur in the action of a trigger, variable instantiations can be passed from the condition to the action part of the trigger. Sometimes it is possible to refer to the modified data or to the previous database state within a condition.

The *action* of a trigger specifies the operations which have to be performed if the event occurs and the condition is satisfied. In analogy to events, actions may be data modifications in the relational database context, and method invocations in the object-oriented database context. Often database operations like "rollback" or any messages are allowed as well. The particular actions possible depend on the specific systems.

It is often possible to specify priorities between different triggers, or to deactivate and activate specific triggers. Some systems permit the specification of the kind of coupling between the occurrence of the events and the evaluation of a trigger's condition, as well as the coupling between the evaluation of the condition and the execution of the action. This allows the designer to indicate, for example, whether the condition should be checked at once when the event occurred, or at a certain point in time (e.g., the end of the transaction). It also allows us to indicate, whether the action should be executed as soon as the condition is satisfied, in extension to the current transaction, or whether the action should be executed at the end of the current transaction, either in extension to it or within a new transaction.

As will be shown in sections 3 and 4, which contain various system descriptions, different ADBSs vary considerably in the features they offer.

2.2 Trigger processing

Firing or *activating* a trigger means executing the action of the trigger. In the database context the activation of triggers is always performed "set-oriented", meaning that the action is executed for the set of instantiations which satisfy the condition, and not only for a particular instantiation as in AI.

The notion of a *transaction* is important for trigger processing in database systems. Operations are grouped into transactions which are considered as atomic units, for concurrency and recovery purposes. Atomicity means that either all operations in a transaction are executed or none of them. Applying a transaction to a database state defines a new database state. Transactions can commit or abort, in the latter case this means undoing all its effects.

Trigger handling in transactions varies across ADBSs. In some systems a new transaction can be started within the action of a trigger, in others the initial transaction can only be extended by actions of triggers, and some systems allow nesting of transactions. Different systems adopt different approaches to the activation of triggers relative to an event:

- *Immediate activation.* Triggers are activated immediately after the event occurred;
- *Deferred activation.* Triggers are activated at the end of the transaction where the event occurred and the transactions are extended by actions of triggers;
- *Separate activation.* Triggers are activated at the end of the transaction where the event occurred and a new transaction is started for the action of a trigger.

In Postgres, triggers are activated immediately after the event has occurred and in the same transaction. Starburst executes triggers at the end of a transaction, and extends the transaction. HiPAC supports different options and explicitly allows a choice between immediate, deferred and separate activation.

Trigger processing determines how events, conditions and actions interact. Processing may be invoked when just one or several events occur. Some systems invoke trigger processing after single events, some after a set of events, some only after transactions. This is also called the *granularity of trigger processing*. We call *tuple-oriented trigger processing* when trigger processing is invoked immediately after any event, and *set-oriented trigger processing* when trigger processing is invoked after sets of events. We now describe these different approaches.

2.2.1 Tuple-oriented trigger processing

Tuple oriented trigger processing is initiated immediately following modification or access of individual tuples or objects. For relational systems this means that after a tuple is manipulated or retrieved, trigger processing is invoked. This may lead to a recursive trigger processing if a trigger is activated and the action of this trigger acts as an event for another trigger. Whenever an event occurs, the corresponding triggers are invoked immediately, see e.g. Postgres. For object-oriented systems the triggers attached to a method are processed immediately after the method is invoked. We will call this approach the TUPLE-OR-APPROACH.

2.2.2 Set-oriented trigger processing

Set-oriented trigger processing is initiated after the modification of sets of tuples or objects. Here the notion of a *transition* is important. A transition is a database state change resulting from the execution of a sequence of data manipulation operations, which may be collected in a transaction. Triggers typically consider only the *net effect* of transitions; to compute the net effect, updates are interpreted in the following way: (1) if a tuple is updated several times, only the composite update is considered; (2) if a tuple is updated, then deleted, only the deletion is considered; (3) if a tuple is inserted, then updated, this is considered as inserting the updated tuple; (4) if a tuple is inserted, then deleted, this is not considered at all. For trigger processing only the accumulated net effect of updates is considered. Actions of triggers may extend transitions.

- **The recognize-act-cycle**

The recognize-act-cycle is a classic AI method of rule execution. In databases there is an initial phase determining those triggers which may be activated as a consequence of the net effect of events and whose conditions are satisfied. This set of triggers forms the first "conflict set". After this initial step the following operations are carried out: (1) perform the conflict resolution and choose one trigger out of the conflict set; (2) execute the statements in the trigger's action (this is called the "act phase"); and (3) determine the next conflict set (this is called the "match phase").

The cycle repeats until there are no more triggers eligible to run. We will call this approach the *RAC-APPROACH*.

- **The transition approach**

In this approach, the database state change resulting from a user transaction creates a first transition. A trigger is now considered for activation if at least one of its events occurred in the net effect of the transition. In contrast to the RAC-APPROACH, the conflict resolution is applied immediately at this point, before the conditions are checked. Next, the condition of the chosen

trigger is checked. If the condition cannot be satisfied, another trigger is chosen, otherwise the action is executed. After execution of the action, all triggers not yet considered only remain under consideration if their event occurred in the composite transition created by the initial transition and the subsequent execution of the last trigger's action.

Trigger processing terminates when there are no more triggers under consideration, or when there is no trigger whose condition is satisfied. During trigger processing, triggers already considered are considered again if their event occurred in the composite transition since the last time it was considered. If it has not yet been considered, it is considered if the event occurred in the transition since the start of the transaction. This approach is, for example, adopted in Starburst. We will call the method the *TRANSITION-APPROACH*.

- **The event approach**

Trigger processing can also be set-oriented and based on a sequence of events E_i, $i = 1, \ldots n$. In contrast to other approaches, *all* triggers which have to react to an event are activated first, before the next event is considered. For an event E_i, corresponding triggers are activated if their conditions are satisfied, possibly considering priorities. These can of course change the database and hence produce new events. If event E_i is "saturated", i.e., all triggers were activated which correspond to this event, then the next event E_{i+1} is considered.

Trigger processing stops when the database does not change any more or when no further trigger can be activated. This method of processing can be used for immediate or deferred activation of triggers. Simon et al. (1992) use the approach in implementing triggers on top of a relational database system. We will call this approach the *EVENT-APPROACH*.

- **Trigger processing based on production compilation networks**

A different approach to trigger processing is to compile them into a Petri-net based execution model, a so-called Production Compilation Network (PCN) (Barthez et al., 1988). Basically, a PCN records the relationship between triggers and relations stored in the database. Each trigger is represented by a transition linked by input edges, with all relations appearing in the trigger condition, and by output edges, with all relations appearing in the action expression (note that "transition" is used here in a different context than in combination with the net effect). Furthermore, these transitions are labelled with the conditions that have to be met.

The execution phase for a PCN corresponds to the recognize-act cycle. First those transitions are determined which match the current database state, and one transition is chosen according to some conflict resolution strategy. The evaluation of the transition is performed by issuing relational algebra expressions to the database system. The evaluator returns a set of elements matching the condition which are used to "fire" the transition, and thus update the database according to the triggers actions. For a detailed description of the semantics of a PCN see Kiernan et al. (1987).

The execution of the PCN corresponds to the firing of the transitions. Trigger processing stops when no more conditions are satisfied, or when no new database state can be reached. The activation of a network corresponds to a forward execution of the corresponding triggers. We will call this approach the *PCN-APPROACH*.

2.3 Conflict resolution strategies

If several triggers can be activated at the same point in time, a conflict resolution strategy has to be applied to solve the conflict of choosing just one trigger. If no mechanism for conflict resolution is provided then one trigger is usually chosen randomly, or all of them are activated. Postgres and Ariel allow user-defined numeric priorities to be attached to triggers. Starburst allows us to give one trigger t_1 priority over another trigger t_2 by defining "$t_1 > t_2$". HiPAC supports concurrent execution of triggers in separate transactions. RDL1 permits procedural control through definition of control sequences on the user level. Ariel uses different criteria, e.g., it first tries to select the trigger with the highest priority, or the trigger whose condition is the most selective one.

Agrawal et al. (1991) give data structures and an algorithm for implementing a priority system permitting incremental combination of user-defined priorities with default priorities (e.g., the oldest trigger first). Triggers are considered in the default order unless user-defined priorities redefine the order. Using this system, trigger processing is repeatable and deterministic, i.e., for a given set of triggers and priorities, the triggers are considered for execution in the same order if the same transaction is executed twice on the same initial database.

2.4 Termination and determinism of trigger processing

In ADBSs trigger behaviour can be complex and unpredictable, even for a small set of triggers (Zhou & Hsu, 1990; Hellerstein, 1991; Aiken et al., 1992). Complex interactions between triggers may make it difficult to maintain and extend the systems, or to predict the behaviour, so a precise execution semantics is essential. Trigger processing can also be subject to infinite loops, due to the property that trigger actions may cause state changes, which lead to the activation of further triggers or even the same trigger again. Thus, problems for trigger processing are

- termination, i.e. whether trigger processing will terminate or loop, and
- determinism, i.e., whether a unique final database state can be reached for a set of triggers, a given database state and some events.

There are some algorithms for identifying triggers which might cause problems with respect to termination and determinism. Aiken et al. (1992) give static analysis methods for determining whether arbitrary sets of triggers are guaranteed to terminate, and if so whether they produce a unique final database state. These algorithms may not always detect when a trigger set satisfies the above properties, but isolate triggers which cause problems and determine criteria which might solve the problem, e.g., in terms of priorities between triggers.

Zhou and Hsu (1990) specify the class of "interference free" triggers as being deterministic, and presents conditions to check whether triggers are interference free. In extension to this, Hellerstein (1991) specifies the class of "implied-interference-ordered" (iio) triggers, and demonstrates that they are deterministic. The class of iio triggers is defined based on two relationships between triggers, which are used to build a dependency graph: an "implies" relationship when the action of a trigger P causes another trigger Q to be fired, and an "interference" relationship when the activation of a trigger P prevents the firing of another trigger Q. If triggers are not interference free, then conflict resolution has an influence on the fixpoint that will be reached from any state of the database. Consequently, a consistent ordering has to be ensured for interferences and also for implied interferences to ensure determinism (for details see Hellerstein, 1991).

Also Kiernan et al. (1990) investigate the problems of determinism, in the context of Production Compilation Networks, and propose "control predicates" to guarantee that a network has a unique fixpoint.

Other systems simply include a depth limit for trigger processing as a run-time solution.

3 Triggers in relational database systems

Relational database systems (RDBSs) have become very popular because of their simple underlying concepts. To increase the functionality, RDBSs have been (amongst other extensions like abstract data types) extended with active functions (ADBSs), while still keeping standard database functions like transaction handling, recovery, and so on. ADBSs based on RDBSs include Postgres, Starburst, Ariel and RDL1.

3.1 Postgres

Postgres (Stonebraker et al., 1989; Stonebraker & Jhingran, 1990) is the successor of the RDBS INGRES (Held, 1975). In order to support new applications better, the design goals for Postgres

included support for complex objects, user-definable data types and operators, and triggers, while making as few changes as possible to the relational data model. Postgres proposes support for complex objects by supporting an extensible type system so that newly defined types can be used for the definition of columns of relations. New operators on these columns and new access methods can also be defined. The query language POSTQUEL is based on the INGRES query language QUEL. We concentrate here on the trigger facility of Postgres.

3.1.1 Trigger syntax

In Postgres triggers consist of an event, a condition and an action. A trigger is defined by a "define rule" command with the following syntax:*

```
define [instance/rewrite] rule <trigger-Name1>is
on <event> to <object> [[from <POSTQUEL-clause>] where <POSTQUEL-
clause>]
then do [instead] <action>
```

The keywords *instance* (resp. *rewrite*) indicate which implementation ("*instance* level" or "query *rewrite*" implementation, see below) should be applied. Default is the instance level implementation.† An *event* can be one of the database operations "retrieve", "replace", "delete" or "append". *Object* refers either to a relation name or to columns of relations. The *FROM and WHERE clauses* are normal POSTQUEL FROM and WHERE clauses without modifications to the database. The action part of the DO clause is a collection of POSTQUEL commands where "new" or "current" can be used in combination with a column of a relation, thus referring to the new or current values. An *action* is a POSTQUEL command, or the keyword "nothing". The flag *instead* will perform an action instead of the event. Without this flag the action is executed in addition to the event. Thus it is, for example, possible to hide information.

Assume, for example, a relation *EMP(name, salary, dept)* and the trigger:

```
define rule example_1
on retrieve to EMP.salary where EMP.dept = "shoe"and user() = "Joe"
then do instead retrieve (salary = null)
```

Now, whenever Joe wants to access the salary of somebody of the shoe's department, he will get the answer that the salary is "null", instead of retrieving the actual value.

It is considered to allow numeric priorities (integers in the range of 0–15) to be attached to triggers. In case of conflicts at run-time the trigger with the highest priority should be executed first. If several triggers have the same priority, one is chosen randomly.

3.1.2 Trigger processing

Postgres follows the TUPLE_OR-APPROACH. When an individual tuple is accessed, updated, inserted or deleted within a transaction, the appropriate triggers are immediately instantiated and executed. After any (user- or trigger-generated) tuple modification, candidate triggers are identified, the conditions of the triggers are checked, and if the conditions are satisfied, the corresponding actions are executed. If the action is "rollback", trigger processing terminates and the transaction is aborted. The action of a trigger belongs to the same transaction as the user command that caused the activation of the trigger. It is not allowed to start a new transaction.

The following example trigger adjusts Sam's salary whenever Joe's is changed:

```
define rule example_2
on replace to EMP.salary where current.name = "Joe"
then do replace EMP(salary = new.salary) where EMP.name = "Sam"
```

*Optional clauses are surrounded by square brackets.

†The instance level implementation is sometimes referred to as tuple level implementation.

Whenever Joe receives a salary adjustment, the event is satisfied and Joe's current tuple and proposed new tuple are available to the execution routines. His new salary is substituted into the action part of the trigger which is then executed.

Triggers can be activated in an *early or a late evaluation mode*; this is intended to be determined by an optimizer. In early evaluation the event awakens the trigger and causes execution of the action part. In late evaluation the action part is not executed until there is a read request on the data which has to be modified, i.e., Sam's salary in the above example.

There are two further processing modes in Postgres: the *instance level* and the *query rewrite processing*. The instance level processing puts locks on tuples of relations, the query rewrite processing is implemented as a module between the command parser and the query optimizer. For the *instance level implementation*, information about the trigger is recorded in the system catalogs and "locks" are placed at either the relation or the instance level. Three types of locks exist: "event", "import" and "export" locks (for details see Stonebraker et al., 1988 and Stonebraker & Jhingran, 1990). Briefly, event locks are placed on columns of tuples appearing in the event specification of a trigger. When an event occurs on a tuple which has an event lock, the condition of the trigger is checked. Event locks thus identify the trigger that must be awakened when a particular event occurs. Whenever updates are performed on the database, the locks have to be updated as well. The following example shows how event locks are placed:

```
define rule example_3
on retrieve to EMP.salary
where EMP.age > 50 and EMP.dept = DEPT.dname and DEPT.floor = 1
then do instead retrieve (salary = 10.000).
```

Event locks for this trigger are placed on the column "salary" of all employees older than 50 years and working in a first floor department.

Instance level implementation is efficient if events are restricted to a small number of tuples. To improve performance for triggers which involve complete relations in their event parts, the *query rewrite implementation* is used. This method basically takes the query and the trigger and rewrites the query, using the information of the trigger, into a form which can be evaluated efficiently. This is performed by a module between parser and query optimizer which processes incoming Postgres commands. For a trigger such as

```
define rule example_4
on retrieve to TOY_EMP
then do instead
  retrieve (EMP.OID = EMP.OID, EMP.name, EMP.age, EMP.salary)
where EMP.dept = "toy"
```

and a query like

```
retrieve (TOY_EMP.salary) where TOY_EMP.name = "Sam"
```

it is more efficient to rewrite the query, using the trigger, to the following new query:

```
retrieve (EMP.salary) where EMP.name = "Sam" and EMP.dept = "toy"
```

instead of using the instance level implementation which would mark all tuples in TOY_EMP.

To summarize, if the event covers most of a relation, then query rewrite may be most suitable; if only a small number of tuples is involved, the instance level implementation may be preferred.

3.2 Starburst

The Starburst project was initiated in 1985 at the IBM Almaden Research Centre. The goal of the project was to build an extensible DBS prototype that can support non-traditional applications and

serve as a testbed for future database technology (Widom et al., 1991; Widom & Finkelstein, 1990). A complete DBS should be built that is extensible at every level: for data storage methods, access methods, execution strategies, query optimization, query analysis, query language, and so on. For example, at the data storage level each table may be stored in any one of several storage methods, and for each storage method there is a set of operations available. The Starburst group chose to build an extended RDBS rather than an OODBS, because of the popularity and acceptance of the RDBSs.

Extensions of Starburst support various advanced features, including hierarchies of user-defined types and functions, large unstructured and structured complex objects, and triggers to respond to changes in the database (Lohman et al., 1991). Trigger processing is fully integrated into the database query and transaction processing (including features like rollback or recovery).

3.2.1 Trigger syntax

The Starburst trigger language is based on SQL. The syntax is:

```
create rule <Trigger_Name> on <relation-name>
when <transition predicate>
[if <condition>]
then <action>
[precedes <trigger-list1>]
[follows <trigger-list2>]
```

Trigger events include "update", "insert", or "delete" database manipulation operations. They have to be specified explicitly by the user in a *transition predicate* in combination with columns of *relation-name*. Several events can be specified in Starburst in contrast to other systems. The *condition* is specified by a relational expression (using SQL), the *action* by a sequence of SQL data manipulation operations. The condition and action parts may refer to temporary relations, transition tables, recording the insertions, deletions, etc. that have occurred during a transition, and thus may refer to both new and old values. Triggers can be explicitly activated and deactivated, and the keywords *precedes* and *follows* can be used to specify priorities among triggers.

The following example causes the rollback of a transaction whenever an updated or inserted salary exceeds 150000:

```
create rule sal_extrem on emp
when (inserted, updated(salary)),
if "exists (select * from i as (inserted()) where i.salary > 150000) or
exists (select * from nu as (new_updated()) where nu.salary > 150000)",
then "rollback"
```

3.2.2 Trigger processing

Starburst trigger processing follows the TRANSITION-APPROACH; it is set-oriented and only the *net effect of transitions* is considered (section 2.2.2). Unlike HiPAC and RDL1, the net effect is computed separately for every trigger.

Triggers are activated at "assertion points". There is an assertion point at the end of each transaction, and additional user-defined assertion points will be definable within transactions. For trigger processing, the Starburst "attachment mechanism" permits definition of procedures which are called after each operation on a tuple of a relation (Widom et al., 1991). A logging mechanism for recording trigger firings, and a conflict resolution strategy based on priorities attached to triggers by the user are also provided by Starburst (Agrawal et al., 1991) (see section 2.3).

To summarize, considerable work has been devoted to trigger processing in the Starburst system. The processing model is based on techniques for computing the net effect of transitions, and integrated into database transaction handling.

3.3 Ariel

Ariel (Hanson, 1989, 1992) is built with the EXODUS database toolkit and integrates a RDBS with a trigger system. Ariel focuses on integrating the trigger system with transaction processing and on efficiency. Efficiency is addressed by a method of condition testing based on an extension of the Rete network technique developed in AI (see section 7.1). The algorithm implemented in Ariel is called A-TREAT, and is an optimization of the original Rete algorithm for the database context. The query language of Ariel is a subset of POSTQUEL extended with a trigger language.

3.3.1 Trigger syntax

Triggers in Ariel have the syntax:

```
define rule <trigger-name>
[priority <priority-value>]
[on <event>]
[if <condition>]
then <action>
```

For each trigger a unique name *trigger-name* is required. The "on clause" specifies an *event* and is optional. An event can be a data manipulation operation like "append to", "delete from" or "replace to", in combination with a relation name. Ariel *conditions* are POSTQUEL queries, and *actions* can be any data manipulation or data definition commands. Conditions can refer to the old database state (e.g., to compare tuples with those in the new database state). Optionally, either the event or the condition can be omitted. Triggers having only an "on" clause are intended to fire once at the end of the command in which the event occurred, while having only an "if" clause are supposed to fire if the condition is satisfied after an update.

Priorities are numerical values between −1000 and +1000 which can be assigned to triggers. The default priority is *0*.

3.3.2 Trigger processing

Trigger processing is performed after each Ariel "top-level command", which is not nested inside any other command. Thus the user is able to vary the granularity of trigger processing: a single data modification command can initiate trigger processing, or commands can be grouped into a transaction which then indicates trigger processing.

Trigger processing is based on the "recognize-act-cycle" (RAC-APPROACH). As with production systems, the expensive step of the cycle is the "match step", where those triggers have to be determined whose conditions are satisfied. The A-TREAT variant of Rete is used to perform this efficiently. The inputs for the network are those database tuples which have been modified. Tuples which pass the network represent conditions which are satisfied.

For the act phase set-oriented activation of the trigger is performed, with the action executed for the set of all tuples which passed the network, and which are temporarily stored in the database.

Actions of triggers extend the transaction which initiated trigger processing. As in Starburst trigger processing takes into account the net effect of data manipulation operations.

If numeric priorities are assigned to triggers they do not need to be unique. In case of conflicts Ariel chooses one trigger according to the following criteria: (a) First try to select the trigger with the highest priority; (b) if there are still conflicts, select the trigger whose condition is matched recently by changes; (c) select the trigger whose condition is the most selective one; else (d) select an arbitrary trigger.

3.4 RDL1

The RDL1 system is a rule-based programming environment which is integrated into a RDBS based on the SABRINA system (Kiernan et al., 1987, 1990; Barthez et al., 1988). The data model

of RDL1 is an extension of the relational model with user-definable data-types and operators. User-defined types can be described in an "is-a" hierarchy and inherit operators. The RDL1 rule language is an extension of Datalog (Ceri et al., 1989) and supports, for example negation, updates and complex domains. RDL1 uses triggers to implement a deductive database system.

3.4.1 Trigger syntax

The RDL1 rule language consists of a condition part *Condition* which is a relational calculus expression, and an action part *Action* which is an arbitrary number of insertions and deletions of tuples into base relations:

IF Condition THEN Action

Relational calculus expressions permit combinations of predicates with logical connectors like "and", "or" and "not". Predicates can be either base relations or comparison operators. A range restriction condition requires that all the variables appearing in the action part have to appear positively in the condition part.

The following example computes the classical ancestor relation from a base relation "parent":

IF parent(X, Y) THEN + ancestor(X, Y)
IF parent(X, Y) and ancestor(Y, Z) THEN + ancestor(X, Z).

3.4.2 Trigger processing

Trigger processing follows the PCN-APPROACH. RDL1 programs are compiled into an internal execution model called Production Compilation Network (Gardarin et al., 1989) which is stored in the relational database. The PCN is the basis for the integration of the language with the DBMS and also for query optimization techniques.

The activation of a network corresponds to a forward execution of the corresponding triggers. Inference is performed using forward chaining by firing transitions. If several transitions can be fired conflict resolution is applied. When no transition is enabled, the network has reached a stable state. When the evaluation of the condition returns a set of matching instantiations the action is executed for all the instantiations of the condition. A sequence of actions is not interpreted as a sequential execution of the elementary actions but as one atomic database update that maps the database state into a new state. The effect of a trigger is determined by the effect on the database state. Only those triggers which are effective are fired.

Problems with infinite loops arise when deletions occur in the action part of a trigger, as a trigger t_1 can insert tuples which are deleted by another trigger t_2 and inserted again by trigger t_1. RDL1 proposes "control predicates" that do not permit triggers to be fired multiple times with the same tuples (for details see Kiernan et al., 1990).

The conflict resolution strategy adopted in RDL1 is based on stepwise saturation and on a procedural control facility on the user level. An implicit partial ordering is introduced among the triggers in the conflict set. If two triggers t_1 and t_2 are in the conflict set, then $t_1 \leqslant t_2$ iff there is a path in the PCN going from t_1 to t_2. This means that the content of a relation must reach a stable point before considering another trigger using this relation. Further, control structures are provided for the user to define an explicit ordering between triggers. These can describe sequential or non-deterministic execution, or combinations, e.g., $SEQ(\ SEQ(t_1, t_2\ BLOC(t_3, t_4))\ t_5)$, where t_i are trigger names, *SEQ* denotes the sequential execution of triggers, and *BLOC* a non-deterministic execution. A control string is evaluated from the innermost parenthesis to the outermost. If none of these strategies can be applied a trigger is chosen randomly from the conflict set.

3.5 Transforming a passive DBS into an active DBS

In this section, two systems are presented which aim to provide an architecture for transforming a passive DBS into an ADBS: The Alert system suggests a layered architecture supporting a trigger language, while an extension of RDL1 suggests coupling a Trigger Monitor with a RDBS.

3.5.1 Alert

Alert (Schreier et al., 1991) is implemented in Starburst. It introduces the notion of *active tables* and *active queries*. Active tables are append-only relations in which the tuples are never updated and new tuples are added at the end. Active queries are queries that range over active tables. Active queries support queries on data which may be added to the database in the future, as well as current data.

- **Trigger syntax**

Alert does not introduce a new syntax to define triggers. Triggers correspond to active queries where the *condition* is specified by SQL FROM and WHERE clauses, and the *action* is specified in a SELECT clause. An action can be any SQL data manipulation statement, a predefined function or program. The definition and manipulation of an active query is identical to standard SQL queries.

The following trigger *cost_watch*, for example, sends a mail to "Irv" using a function "sbmail" whenever an expense claim exceeds 1000:

```
Create rule cost_watch as
SELECT sbmail("Irv", ename)
FROM journal
WHERE method_name = "expense_claim" AND expense_amount > 1000
```

This trigger is specified as an active query over the relation "journal". The condition is specified in the WHERE clause, the action of invoking "sbmail" is specified in the SELECT clause. This action is performed for each tuple selected by the query.

Triggers can be activated and deactivated. The user can also define the *transaction coupling* mode, and the *assertion mode* for trigger processing. The transaction coupling mode ("separate" or "same") specifies whether the action should be executed in a separate transaction or not. The "immediate" assertion mode means that a trigger is activated as soon as the trigger condition is satisfied (in contrast to "deferred" assertion mode).

- **Trigger processing**

A so-called "Alert monitor" manages active tables and monitors changes to the database state. The management of active tables differs from the management of normal relations only in the case of insertions. Again, the Starburst attachment mechanism is used to send messages to the Alert monitor whenever relations are updated. Alert follows the TUPLE-OR-APPROACH. Whenever a trigger is activated, Alert opens the active query associated with it. After an update of the respective active table, the Alert monitor checks the condition of all active queries defined on this relation and gives the list of qualifying active queries to the conflict resolver, which determines the order of execution. The conflict resolver chooses a trigger, and the qualifying tuples are passed to the action. Processing of a trigger stops when there are no more qualifying tuples, then the conflict resolver chooses the next trigger.

3.5.2 RDL1 extension

Simon et al. (1992) describe how to couple a so-called "trigger monitor" with a RDBS. Instead of integrating triggers with a DBS, they adopt a "loose coupling" approach to make their method applicable for any RDBS. This, and the fact that deductive rules (which are implemented using triggers) are considered as well, distinguish the work from other systems. The system represents an extension to the deductive database language RDL1 (see section 3.4). It is formally described by means of a partial fixpoint operator, which integrates both the deductive database and active database paradigms.

- **Trigger syntax**

A trigger consists of an if-then statement

```
<name> is
```

```
if <condition>
then <action>
```

where the *condition* is a tuple relational calculus expression, and the *action* is a set of elementary actions e.g., database updates, variable assignments, or procedure calls. *Name* is the name of the trigger. Triggers are encapsulated in modules which collect logically related triggers. Priorities can be assigned to triggers as well. Triggers can be used in two ways:

1. to deduce data (in this case they react to the event "select"), e.g., which persons are rich
   t1 is if person (X) (X.income > 500000) then + rich(X)
   Deduced relations are stored only temporarily.
2. to modify the database as reaction to external events e.g.,
   t2 is if emp(X) and not exists Y in dept (X.dept_no = Y.dept_no)
   then − emp(X).

Thus there is a single notion of triggers whether used to deduce data or to react to events.

Triggering events need not be specified explicitly but are implicit in the trigger definition and are derived by the system at compile-time. Events are database operations such as "select", "insert", "delete" or "update".

Activation of triggers can be in *immediate* or *deferred* modes, which are always specified by the user in the module. Triggers which deduce data are always activated immediately.

- **Trigger processing**

Trigger processing is part of the execution of a given transaction. For trigger processing the EVENT-APPROACH is adopted. First the sensitivity of a module with respect to an event is tested. If a module is relevant then the PCN data structure is built in main memory, and triggers are evaluated using this structure. In case a module is considered repeatedly for execution, the system checks whether the database state has changed since its last execution. If not, the system considers the next module. Modules are computed one after the other, and each module is computed up to saturation before exceeding the next one.

The Trigger Monitor is an executable program which results from the compilation of the modules. It automatically activates and executes triggers depending on the actions taken by an application program, and is coupled with the RDBS. Coupling is achieved using communication software. Communication of an application program with a DBS is then established by the Trigger Monitor. (Note: redefinitions of triggers imply that the Trigger Monitor has to be recompiled.)

3.6 Overview

Postgres, Starburst, Ariel and RDL1 illustrate different approaches to integrating a trigger functionality into a RDBS. Triggers in Starburst follow a set-oriented semantics, trigger processing is deferred to the end of a transaction, and extends the transaction. In Postgres and Alert trigger processing is tuple-oriented. In Ariel emphasis is put on efficient implementation of condition testing based on an optimization of the Rete network. In contrast to the other systems, the RDL1 system incorporates triggers to implement a deductive database system. Alert and an extension to RDL1 demonstrate how a passive DBS can be transformed into an ADBS.

4 Triggers in object-oriented database systems

Some work has been done on the integration of object-oriented concepts and triggers. For example Beeri and Milo (1991) or Diaz et al. (1991) study the integration of the paradigms of ADBSs and OODBSs. Object-oriented concepts like encapsulation, inheritance, classes, objects and methods are supported. Kim et al. (1992) give a framework for supporting triggers in OODBS and explain the trade-off between performance and flexibility in implementing and using triggers. This section describes HiPAC, Ode and Tanguy.

4.1 HiPAC

HiPAC offers the most sophisticated design of the various systems considered, however the implementation is still in main-memory and not yet fully integrated into a DBS. The HiPAC project has investigated the efficient management and evaluation of triggers in an object-oriented environment (McCarthy & Dayal, 1989; Chakravarthy, 1989; Rosenthal et al. (1989). Triggers, which are called "event-condition-action" rules, are first-class objects: there is a class "rule", every trigger is an instance of this class, and attributes and operations are defined on this class.

4.1.1 Trigger syntax

In HiPAC a trigger is defined as an instance of a class "rule". The trigger syntax is

```
Event: <event>
Condition: <condition>
Action: <action>
```

where

- An *event* is an expression of an event algebra. An event can be a primitive event or a combination of primitive events according to the event algebra (Chakravarthy, 1989). Primitive events are generic database operations, temporal events, external notifications (by other application programs), creation of objects, or operations on instances of objects.
- A *condition* is a query in extended relational algebra. Results of a query can be passed to the action.
- An *action* is an arbitrary expression or program. In contrast to other systems HiPAC allows calls to application programs within an action.

An example of a trigger is

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

which orders an item if the inventory falls below a threshold value for that item.

In addition, triggers have the attributes:

- *E-C coupling* which specifies when the condition has to be evaluated relative to the transaction in which the triggering event occurred, and
- *C-A coupling* which specifies when the action is executed relative to the transaction in which the condition is evaluated.

For these attributes one can choose between "immediate", "separate" and "deferred" coupling with the following meanings:

- *immediate*: when the triggering event occurs (the condition is satisfied), evaluate the condition immediately (execute the action) in the same transaction.
- *separate*: when the triggering event occurs (the condition is satisfied), evaluate the condition (execute the action) in a separate transaction.
- *deferred*: evaluate the condition (execute the action) at transaction commit time in the same transaction.

The operations which are defined for the class "rule" are

- *Fire*: Evaluate the condition and, if it is satisfied, execute the action.
- *Disable*: Disable automatic triggering.
- *Enable*: Enable triggering.

Further, it is possible to create, delete or modify a trigger.

4.1.2 Trigger processing

The HiPAC execution model emphasizes concurrency, and is based on nested transactions and assignment of conditions and actions to different transactions based on the pre-defined coupling modes. A nested transaction is a transaction which is wholly contained in another transaction. The effect of a subtransaction becomes permanent when it and all its ancestors commit. When a transaction aborts, its effect and the effects of its descendants are undone.

In HiPAC an "event detector" reports the occurrence of primitive events to a "trigger manager". The trigger manager is responsible for the processing of triggers, their maintenance and optimization. When an event is detected, the trigger manager identifies relevant triggers, calls the transaction manager to create necessary subtransactions, and passes the triggers to be evaluated to the condition evaluator. The "transaction manager" implements the nested transaction model, starts and terminates transactions and is responsible for concurrent control. It also acts as an event detector (e.g., the termination of transactions) and reports to the trigger manager. No conflict resolution strategy is applied. HiPAC activates triggers concurrently or serially.

HiPAC uses a data-structure called a *signal graph* for efficient condition checking. A signal graph is a directed acyclic graph where leaf nodes represent either signal relations or base relations (for detail see Chakravarthy, 1989). The event and condition of a trigger is translated into a signal graph.

HiPAC extends the relational algebra used for expressing queries and manipulation of data with an operator *changes* to express conditions involving changes to stored and derived relations. To support the notion of changes, delta-relations are introduced reflecting the difference between two database states, as well as an incremental implementation of the relational operators "join", "select" and "project". An operator "changes" expresses how a derived relation changes when one of the input relations changes. The output of the changes operator is a delta-relation expressing the net changes produced in that derived relation.

In contrast to other systems, the HiPAC approach is quite sophisticated for specification of events, coupling modes, and incremental evaluation of conditions. The various coupling modes permit control of transaction creation and nesting, and the evaluation of conditions and execution of actions relative to the transaction in which the event occurs. If several triggers can be fired, HiPAC executes the different actions concurrently defining new transactions for each action.

4.2 Ode

Ode (Gehani & Jagadish, 1991) is an OODBS providing facilities for constraints and triggers. Both are specified declaratively and are associated with class definitions which can be inherited. Constraints are triggered implicitly by any invocation of any method in the respective class, but triggers have to be activated explicitly by a command, which specifies the object for which they are activated. Once activated on an object, a trigger is then implicitly activated by any invocation of any method on that object. Constraints are mainly used to ensure consistency between objects, with the consequence that transactions are aborted in case of inconsistencies. All objects of the same class have the same constraints and these constraints must be satisfied by all objects, but different triggers may be activated for different objects even when the objects belong to the same class.

4.2.1 Trigger syntax

Constraints and triggers are specified together with the class definition. The syntax is

```
condition: action
```

where *condition* is a boolean expression referring to components of the class, and *action* is a statement that is executed when

- the condition is *not* satisfied in case of constraints, and
- the condition *is* satisfied in case of triggers.

If no action is given, the transaction is aborted.

Constraint checking can be performed immediately after the object was accessed by a method, or it can be deferred to the end of the transaction. Therefore constraints are distinguished according to whether they are *hard* or *soft*. Soft constraints are checked at the end of a transaction, in an unspecified order. Hard constraints are checked immediately after the method was invoked. In addition to hard and soft constraints, Ode offers triggers to monitor the database for conditions. Triggers have to be activated explicitly on a particular object. There are "once-only" and "perpetual" triggers. A once-only trigger is automatically deactivated after it has been fired and must be explicitly activated again; a perpetual trigger is always active. There are also timed-triggers where the action has to be executed within a period of time.

4.2.2 Trigger processing

Constraint and trigger processing is initiated after the invocation of methods. Conditions of hard constraints are checked and the action is executed immediately. When an object with a soft constraint is updated, it is placed in a "to-be-checked" list. The soft constraint is checked only once at the end of the transaction while multiple updates to the same object would cause multiple checks of the same hard constraint. Conditions of triggers are checked immediately, and potential actions are noted for execution after the current transaction, in a separate transaction. This means that aborting the action of a trigger does not result in the rollback of the entire transaction which fired the trigger. As actions of triggers are executed after the triggering transaction, but not necessarily immediately after it, it is possible that the condition which fired the trigger is no more true when the action is executed and it has to be checked again.

4.3 Tanguy

The objectives of Tanguy (Czejdo,1991) are to integrate object-oriented programming with rule-based programming and permanent storage of knowledge in a Smalltalk environment. Knowledge which is specific to a particular application is supposed to be kept in application programs, general knowledge is supposed to be stored permanently in the knowledge base.

Triggers are called "activation pattern controlled rules", and are used for integrity checking and exception-handling, for example triggers are subdivided into *before-* and *after-triggers* depending on if they are activated "before" or "after" a particular message is executed.

4.3.1 Trigger syntax

A trigger in Tanguy has the syntax:

```
trigger-name <Name>
context: BEFORE // AFTER <context>
condition: <condition>
action: <action>
continuation: CONTINUE // RETURN
priority: <integer>
```

The *condition* is a Smalltalk message that returns a boolean result, and the *action* is a sequence of Smalltalk messages. Triggers have a *name* and are only activated in particular *contexts* which are classes of messages, and are subdivided into *before-* and *after-triggers* depending on whether they are activated "before" or "after" a particular message has to be executed. Also two possible *continuations* exist: a keyword "continue" continues trigger processing, and a keyword "return" terminates trigger processing. *Priorities* can be assigned to triggers; in case of conflicts the trigger with the highest priority is activated first. Triggers can also be activated and deactivated.

4.3.2 Trigger processing

Triggers are processed directly after messages. If a particular message is called, the set of before-triggers is computed which have this message in their context, and a forward-chaining inference process is started for these triggers. If this inference process terminates with "continue", the

message is executed; then the set of after-triggers is computed and an inference process is started for these triggers.

Triggers are used for

- integrity checking (e.g., as "before-trigger"): Assume a trigger t_1 which states that students can be enrolled for a course only if they satisfy the acceptance policy of this course, else the message is not executed and the trigger continues with "return".
- exception-handling, e.g., a "before-trigger" t_2 can deactivate the above trigger t_1 for some classes of students, and return with "continue". Then the message is executed, and then an "after-trigger" t_3 activates trigger t_1 again.

Note that if a trigger has an empty context and the flag "after", this trigger is considered to be activated after any message was executed, and the condition is then always checked. It is therefore reasonable to give a context to restrict condition testing.

4.4 Overview

In this section we have described systems integrating trigger functionality into an object-oriented database system. HiPAC offers the most sophisticated approach, with a variety of coupling modes which allow an "immediate", a "deferred" or a "separate" execution of triggers, and an execution model based on a nested transaction model. HiPAC also offers the greatest variety for the event specification with a dedicated event algebra. In Ode two kinds of active rules are offered, constraint and trigger rules. Both are attached to classes. Constraint rules are implicitly triggered by any invocation of any method in the respective class, triggers have to be activated explicitly by a command which specifies a particular object and some parameters. Tanguy offers so-called "activation pattern controlled rules" and use them for integrity checking and exception-handling. The rules are only fired in particular contexts and are subdivided into before- and after-rules depending on if they are fired "before" or "after" a particular message is executed.

Part II: Reactive processing in artificial intelligence

5 History and overview

The use of rules to provide reactive capabilities for databases has come to prominence only quite recently, but closely related techniques have been widely used in the field of artificial intelligence for many years. Newell and Simon (1972) first recognized the potential of production systems as a basis for constructing general programming systems, and by 1978 Waterman and Hayes-Roth could already review work on the use of production systems in user interface software, knowledge acquisition and knowledge engineering, natural language, learning and psychological modelling. Rule-based programming continues to be an important technique in artificial intelligence to the present day.

Newell and Simon were to a great extent responsible for giving production systems their initial prominence. They were interested in their technological potential though the driving force behind their introduction was that Newell and Simon believed that production systems offered a natural basis for building systems that can operate autonomously and adaptively in complex and unpredictable environments. They promulgated both the general idea that symbolic computation provides sufficient means to construct a "general intelligence" and, more specifically, that a simple architecture for interpreting collections of *condition* $\Rightarrow$ *action* rules provided sufficient means for implementation. The appeal of rule-based systems lay partly in their conceptual simplicity and the ease with which rule-interpreters could be implemented, but there were a number of additional features which appeared to make them particularly natural for constructing intelligent "agents":

1 Rules can fire in response to patterns of data and can therefore respond directly to observed states or events in the environment. The ability to respond to unpredictable situations or occurrences is held to be a necessary capability for adaptive behaviour.

2 As with modern deductive systems Newell and Simon argued for the desirability of clearly separating control mechanisms from task knowledge. A fixed architecture for selecting and firing rules provides an adequate control structure for applying a variable body of knowledge expressed in condition-action form.
3 The modularity of a knowledge base or rules was thought to support learning of new knowledge, as condition-action rules could be generated and added to the knowledge base in the light of experience.

However, production systems have a longer history even than that; their origins were succinctly summarized by Newell and Simon (1972):

"The production system was one of those happy events, though in minor key, that historians of science often talk about: a rather well-prepared formalism, sitting in wait for a scientific mission. Production systems have a long and diverse history. Their use in symbolic logic starts with Post (1943), from whom the name is taken. They also show up as Markov algorithms (Markov, 1954). Their use in linguistics, where they are so called rewrite rules, dates from Chomsky (1957). As with many other notions in computer science, they really entered into wide currency when they became operationalized in progamming languages, first in string manipulation systems . . . and SNOBOL . . . and in compiler translation languages."

6 Reactive processing and cognitive psychology

The structure of the rule interpreters developed by Newell and his colleagues in the early seventies (such as PSG: Newell, 1973; Newell & McDermot, 1975) were strongly influenced by ideas about properties of human memory and limited information processing capacity which were then current in cognitive psychology. The distinction between "short-term memory" and "long-term memory" made by psychologists was modelled as a random-access rule memory (which encoded knowledge) and a limited capacity working memory structure (which stored temporary results of processing). The idea that human cognition operates sequentially and has a limited "central processing channel" was the inspiration for development of the "recognize-act" cycle and conflict resolution mechanisms in PSG. These techniques were also incorporated in OPS, a production rules system which came to be widely used in expert systems development (Forgy & McDermott, 1976).

Since Newell's seminal work, many theoreticians in psychology have used active rule systems to model aspects of human cognition, ranging from perception and imagery to language understanding, decision making and learning. Although this work illustrates the flexibility of such methods, it has not generally had technological objectives, and will not therefore be discussed further. However, we shall briefly review two pieces of work which may be of general interest, and point to possible future developments in active rule systems.

Newell's original work using PSG to build small-scale models has evolved into an ambitious international project aimed at the development of an "architecture for general intelligence" (Laird et al., 1987; Newell, 1991). The focus of the work is an active rule architecture called SOAR, which is designed to respond to events in its environment by creating goals, solve them by exploiting a body of knowledge expressed as rules, and to acquire new knowledge as a consequence. SOAR operates in cycles, repeatedly retrieving knowledge relevant to its current goal and making decisions on the results. Decisions are made about the knowledge to use in resolving goals, interpretations to assign data, and operators to apply. If SOAR has relevant knowledge then a goal can be solved directly, but SOAR is more interesting because it has the potential to solve problems for which it does not have specific knowledge. In such cases the architecture detects an "impasse". Impasses cause SOAR to automatically generate new, impasse-specific, goals and recursively decide on problem spaces, states, operators, etc. If the subgoal succeeds the result becomes available in the parent goal context. Following resolution of an impasse the architecture identifies the critical features of the situation and stores this as a new rule for future use.

SOAR demonstrates that a data-driven architecture can implement all the main problem solving strategies that have been developed in AI (including goal-directed strategies). The flexibility of

problem-solving and techniques for learning from successful problem solving may suggest novel techniques for future applications technologies.

SOAR has also played an important role in the work of Cooper et al. (1992), which is aiming to develop an executable specification language for psychological modelling, based on meta-interpretation, logic programming and active rule processing. Their initial system, Sceptic, provides an interpreter for rules of the general form

Trigger: <*Conditions*> ⇒ <*Actions*>

Although closely similar in form to the triggered production rules developed in active database work, Sceptic permits a declarative style of programming and a high-degree of metalevel expressiveness. These attributes are inherited from Prolog, in which it is implemented. The practical consequence of these features are that Sceptic can be used to specify many other active architectures and implement them with surprisingly succinct programs. For example, Sceptic has been used to construct a detailed reimplementation of SOAR. The original program consists of about 100,000 lines of LISP, the Sceptic implementation is about 10% of that, and the core of the SOAR processing engine consists of only 28 generalized productions. A number of other architectures have been similarly implemented. (Sceptic is discussed in more detail later.)

Although the work of Cooper et al. (1992) was motivated by issues in cognitive psychology, it may have implications for engineering active rule systems. The most important observation being metaprogramming capabilities have as much potential in active rule languages as they have been shown to have in logic languages.

7 Technologies for active processing

Following the massive growth of interest in expert systems and knowledge engineering in the nineteen seventies the development of rule-based technology began to diverge from developments in cognitive science. The cognitive science community continued to be concerned with modelling features of human mental processes (e.g., Anderson, 1983; Laird et al., 1987), but the requirements for building practical applications increasingly dominated the development of production system technology and the combination of rule-based processing with other software techniques.

Although many in AI did not share the interest in psychology of Newell and Simon and their colleagues, rule-based architectures were widely taken up. The separation of knowledge from pattern-directed control and modularity of rules had general appeal. In addition, rule-based architectures offered significant potential for parallel, asynchronous and distributed computation. The simplicity of the condition-action representation also opened up possibilities for meta-level reasoning, which were quickly exploited in the explanation and knowledge acquisition facilities of expert system software.

The development of AI tools had to address a number of important practical issues, including efficiency, issues of scale, ease of application engineering and (what we would now call) encapsulation of software functions. Several distinct engineering styles emerged which placed different amounts of emphasis on the rule-based component. Of these the most prominent were rule languages (interpreters and compilers), expert system shells, frame based systems and blackboard architectures. Many of these developments are well known so we only present a brief overview of the points most relevant to the present discussion.

7.1 Languages

The OPS language was originally developed for cognitive modelling, but it came to be used (and commercialized) as a tool for developing expert system applications (notably the R1 system discussed below). Although the basic recognize-act and conflict resolution architecture was kept through successive generations of the language, the syntax and built-in functions of the language acquired more and more features of conventional programming. The language was extended to

include such features as if-then-else, loop processing, arrays and array operations, and so on. Early rule-interpreters were notoriously inefficient, with (at best) cycle times proportional to the product of the size of the rule set and the size of working memory. OPS introduced the Rete pattern matching algorithm which successfully avoids unnecessary recomputation in evaluating the conditions of productions by saving information about partial matches and dynamic indexing of inference data (Forgy, 1982). OPS achieved considerable popularity, partly because of its efficiency but also perhaps because it provided something of a compromise between a pure rule language and a more conventional procedural programming style.

7.2 Shells

The classical expert system shell is EMYCIN, a generalized version of the medical expert system MYCIN (Short, 1976) with all the application specific rules stripped out. EMYCIN supports the development of knowledge bases consisting of rules, but is more oriented towards a particular class of applications, so-called "consultation systems", than a general purpose programming language. EMYCIN introduced a number of techniques based on rule-processing which influenced a generation of expert systems (such as explanations) but departed substantially from a pure, data-driven architecture. Most notably, the basic control structure was goal-driven, using a backward chaining proof procedure, but EMYCIN also supports the inclusion of "forward" rules in the knowledge base to provide a reactive capability if needed. This separation of "forward" and "backward" rules was quite common in later expert system development tools. Typically, backward rules are used to encode declarative inference rules, while forward rules are used to implement procedural computations (e.g., to update the knowledge base or invoke some other non-logical operation). Although the distinction is not necessary in principle it permits execution optimisations and may clarify the structure of the application.

7.3 Frame systems

Among the early knowledge representation schemes used in AI was the semantic network, a simple scheme for defining the semantics of a concept in terms of its relations with other concepts. These evolved into frame systems, which explored a number of ideas later to be adopted in object-oriented databases. A frame can be thought of as a single data structure representing the concept, consisting of a cluster of "slots" for storing property and attribute information. Frames were normally organized into generalization hierarchies, permitting inheritance of properties and attributes from general to specific concepts. Such frame structures were augmented with a variety of mechanisms to carry out inheritance, perform inference and provide reactive capabilities. Reactive functions were provided be *demons* stored in the slots of frames. Demons might be implemented by means of production rules or conventional procedures. Demons might be designed, for example, to detect updates to or retrieval from the slots of a frame and execute a procedure or carry out an inference if the event occurs. These ideas are now familiar in the class structures, inheritance, message passing and encapsulation techniques of object-oriented databases. Their success demonstrated the viability of integrating active, deductive and object-oriented technologies, which is now a focus of much database research.

7.4 Blackboards

Blackboard architectures were developed in AI to support distributed reasoning and to provide a way of organizing and providing communication between distinct processes in large applications. The approach was first developed in the HEARSAY-II speech understanding project (Erman et al. (1980), in which knowledge about low level phonology, morphology, words, syntax and semantics, was organized into separate "knowledge sources". A blackboard system is typically organized as:

- A set of independent knowledge sources that contain the blackboard system's domain specific knowledge. These knowledge sources were frequently implemented as rule-sets, though the model allows other implementation techniques to be used, including mixed techniques.
- The blackboard is a shared data structure through which the knowledge sources communicate with each other by "posting" messages.
- A control system which determines the order in which knowledge sources operate on the contents of the blackboard. The control structure may, for example, reason about such factors as cost of execution and likelihood of achieving a result.

HEARSAY-II and later blackboard systems are of particular interest in the present context because they are particularly well developed for time-oriented computations (such as speech processing). Associated with each knowledge is a set of *triggers*, that specify conditions (data states or the occurrence of events) under which the knowledge source should be activated. If multiple knowledge sources are potentially active a *scheduler* which embodies knowledge about how best to carry out processing in the particular domain, sets priorities. The scheduler is a specialized knowledge source which can be implemented by productions which react to states of, or changes to, the blackboard.

7.5 Logic programming and production rules

It was noted earlier that goal-driven backward chaining techniques for interpreting rules in expert system shells have been augmented with data-driven production rule techniques. It is perhaps surprising that similar augmentation of the goal-driven theorem proving techniques developed in logic programming have received relatively little emphasis. One series of experiments, culminating in the development of Sceptic, shows that this is a profitable direction. This work began with development of a commercial forward chaining expert system shell, XiPlus ((R) Inference Corporation) developed by the Imperial Cancer Research Fund, and later extended to include logic programming capabilities in the PROPS2 shell. PROPS2 also provided triggered rules together with a backward chaining proof procedure for evaluation of rule conditions, augmented by a default reasoning and truth maintenance system to cope with non-monotonic inference. This experiment was successful, but it also suggested that a simpler and more flexible method of combining event-driven computation with and goal-directed proof procedures could be obtained by augmenting standard Prolog with an active rule interpreter.

This lead to development of a production rule extension to Prolog, called Sceptic (Hajnal et al., 1989). A Sceptic program consists of productions of the usual form:

Trigger: <*Conditions*> ⇛ <*Actions*>

but triggers, conditions and actions are standard Prolog terms (relations), which may be first-order. Variables are implicitly universally instantiated, i.e., a production fires for all instantiations of its conditions; there is no conflict resolution. A condition may be any Prolog predicate. An action may be a database update, a further trigger, or some configurable default action.

Sceptic does not of course do anything that cannot be achieved in Prolog, but it provides a valuable syntactic differentiation between procedural and control aspects of an application on the one hand (which can usually be naturally and succinctly implemented as triggered rules), including active control, and the pure (declarative) logical capabilities provided by the Prolog theorem prover.

One of Prolog's great virtues is its meta-level expressiveness which, from a programming point of view, makes it a powerful tool for implementing other language interpreters and computational systems. Sceptic inherits these properties within an active computational framework. Its capabilities have been exploited in a variety of applications ranging from simulation of complex processes and control of image processing and planning systems, to uncertainty management and distributed computation functions. As with the work on SOAR, described earlier, these capabilities are

obtained with surprisingly small programs. The main lesson from the work on Sceptic, which we shall emphasise again below, concerns the benefits of preserving meta-level expressiveness in an active processing context.

8 Active processing in AI: common themes

There were, and still are, significant themes common to the communities interested in natural occurring intelligence and those interested in engineering intelligent automata. Although we have only reviewed a fragment of the work on production systems carried out over the last 20 years, we can identify a number of common themes, which continue to be prominent in the most recent work.

8.1 Pattern-directed computation

The basic idea captured by pattern-directed computation systems is that patterns occurring in data select pieces of code in the system to be activated. Systems organized in this way have been used for both deductive inference and for procedural control. The common structure to most systems consists of three basic components (1) a collection of modules which are "fired" by data patterns, (2) one or more data-structures that may be examined and modified by these modules, and (3) an interpreter or control structure that controls the activation of the modules. Production systems have proved to be among the most flexible and general pattern directed organization. Although the evaluation of conditions and the execution of actions can be arbitrarily complex programs (e.g., Prolog proof procedures in Sceptic; procedures for controlling effector systems in robotics) the most simple form of production system has considerable computational capabilities. The R1 expert system, which was designed to carry out the complex task of configuring computer systems was also a demonstration of the power of this programming paradigm. R1 conditions and actions consisted entirely of matching or recognizing database patterns and adding entries to the database.

8.2 Time-dependent behaviour

A great deal of interest has centred on the application of production rule systems to real-time and event driven behaviour. Work on speech and natural language (e.g., Erman et al., 1980) and signal processing (e.g., Nii & Feigenbaum, 1978) have placed particularly high demands on such capabilities. The combination of symbolic inference capabilities with high-bandwidth real-time operations is difficult but reasonable data-rates are manageable, particularly if low level signal processing is implemented with conventional software (e.g., for data acquisition through sensors, and executing time-dependent control operations for actuators).

8.3 Asynchrony and parallelism

The benefits of modularity in rule-based systems for parallel and asynchronous computation are well known. Parallel proof procedures and constraint evaluation in logic programming depend on the encoding of knowledge in modular rule-based form. Event-driven, production rules continue to be viewed as an important way of organizing knowledge for complex applications: "Autonomous robotic systems designed for hazardous environments require the development of onboard real-time knowledge-based systems capable of generating inference-driven responses to asynchronous external stimulii" (Iyengar et al., 1992). The rules can be organized as a simple set of operationally independent rules or as structures consisting of functionally related rules to provide a basis for both fine-grain or coarse-grain parallelism.

8.4 Pure production systems

Just as work in logic programming emphasized the importance of pure declarative logic programs, much work on production systems has explored pure production systems in which constraints are

placed on the structure of rule antecedents and consequents. Davis and King (1976) suggest that a fundamental characteristic of the pure production systems is a restriction on rule interactions to communication through an open data-structure (e.g., working memory, blackboard) rather than direct calls (transfer of control) between rules. The pure production systems support incremental extension of knowledge bases, simplicity of execution and formal analysis. However, fine-grain encoding of knowledge and execution through side-effects tends to make production systems difficult to program, which is a major reason for departures from the pure model in later engineering work. Structuring of rules into related sets, attaching them to frames or objects, and the introduction of triggers (which provides a way of achieving direct transfer of control) all simplify practical use.

8.5 General programming capabilities

We observed earlier that although the influential OPS system had its roots in psychological modelling, it achieved popularity for practical applications because of its general programming capabilities. Many other production rule languages have followed OPS by enriching the syntax and primitives available for condition evaluation, procedural actions and ability to incorporate foreign functions. Although this has been successful in practice the simplicity of Sceptic, which combines the general programming capabilities of Prolog with the event-driven capabilities of production rules, suggests that the introduction of highly procedural features are not necessary to achieve such flexibility. Pattern-directed control and declarative features of logic programming languages combine to yield clarity, expressiveness and ease of programming for a wide range of applications.

8.6 Meta-level capabilities

The benefits of modularity for meta-level reasoning, whereby programs expressed as rules examine and modify other programs expressed as rules, are also common themes in logic programming and production systems. Work from Vere (1978) and Buchanan & Mitchell (1978) up to the development of SOAR have all exploited the ability of production systems to create new rules in response to events and data-dependencies in the environment. Similar techniques were used to develop capabilities of TIERESIAS, an expert system incorporating knowledge-based knowledge acquisition functions (Davis, 1982). Once again, however, these developments tended to rely on specialized extensions to the rule representation. The experience with Sceptic in implementing SOAR and other architectures suggests that if the underlying language is rich enough (there Prolog) such complications may be unnecessary.

9 Conclusions

ADBSs have recently gained much popularity, as they provide a promising technology for the development of knowledge-base systems by integrating (passive) database technology and AI (active) trigger processing. In this report we have given an overview of reactive processing in Artificial Intelligence and in the database field.

Twenty years of experience with active processing in AI has led to many techniques (e.g., recognize-act cycle, conflict resolution, Rete optimization) now being exploited in current work on active databases. However, fundamental problems for active processing, such as whether processing will terminate or achieve a unique final database state, have not been solved in general. There are also a number of apparent *points of contrast* between the development of active processing in database systems and in AI.

- *Processing*

In AI there is the implicit assumption that during trigger processing only new data can activate a trigger, which means that triggers are implicitly activated by the insertion, deletion or modification of data. On the other hand, trigger languages in the database field usually explicitly mention the

event which might activate a trigger. If the event is mentioned, triggers are called "event-based", if the event is omitted, triggers are called "pattern-based". While in AI triggers react to modifications of a main memory database, the trigger facility was, for the database community, modified to react upon database operations and transitions between database states.

- *Activation*

In the database context activation of triggers is always performed set-oriented, which means that the action is executed for the set of instantiations which satisfy the condition, and not only for a particular instantiation as done in AI (though this is also true in the Sceptic language, and has been found to be a highly desirable feature).

- *State sensitivity*

Although the common use of triggers in database languages yields flexibility, the need to predefine all triggering events places considerable responsibility on the designer to anticipate all the conditions under which data-driven operations will be required. It may be desirable under some circumstances to define triggers which are merely sensitive to states of the database. Of course, production execution can be triggered by a generalized update operation, but this loses the very event-specific operation we are trying to obtain. A compromise may be to provide both state-dependent and event-triggered productions in a single language.

- *Knowledge bases and databases*

It is difficult to establish an in-principle distinction between knowledge bases and databases. However, although there is a considerable convergence between the two kinds of system (e.g., the introduction of is-a hierarchies and the inheritance of operators in RDL1), database technology has tended to emphasize very large data sets with modest numbers of rules, while in knowledge bases the ratio is reversed. Consequently, storage and retrieval efficiency have received less attention in knowledge-based systems research, and the provision of support for complex functions, such as meta-level functions, tend to be viewed as the responsibility of the application programmer by the database designer. The convergence of the two technologies is promising, but the work on active rule systems for databases which has been reviewed experience suggests that active capabilities are currently seen as a limited event-oriented extension to a database rather than having the status of a fundamental paradigm that they have in AI.

- *Uncertainty*

Knowledge-based applications frequently involve uncertainty, and considerable effort has been expended on techniques for representing and propagating uncertainty in knowledge systems. Rule interpreters for expert systems and other AI applications are frequently augmented with mechanisms for revising "degrees of belief", typically expressed as numerical probabilities, or with symbolic methods for automatically maintaining consistency of belief sets as knowledge is updated. This does not yet appear to be a significant feature in work on active database systems.

- *Conflict resolution and rule-prioritizing*

Some of the active database systems reviewed provide support for the applications programmer to attach priorities to triggers (e.g., Starburst, Ariel). Fixed priorities are unusual in AI systems. Conflict resolution methods provide a form of prioritization, but the priorities are set dynamically by the interpreter, based on general heuristics, rather than being statically associated with particular production rules. The relative advantages and disadvantages of these two approaches are as yet unclear, and no doubt different applications will have different requirements.

- *Flexibility of active rules for general programming*

While AI work on pure production systems has discouraged the introduction of procedural syntax and general programming features into rule languages, engineering work in applied AI and in active databases has taken a more pragmatic direction: complex syntax; targetting trigger execution on updates to particular data-structures or tables; transaction oriented operations; time- and date-sensitive operations and mode control. The arguments for such support will perhaps

appeal to database programmers but such specialization should not fatally compromise the ability to write pure applications where there are requirements and where it is desirable to be able to formally analyse, specify and verify the desired behaviour of the application.

Appendix

We conclude this report with a classification of the ADBSs presented according to the design decisions adopted for them.

	RDBS					OODBS		
	Alert	Postgres	Starburst	RDL1	Ariel	HiPAC	Ode	Tanguy
Trigger processing tuple-oriented	X	X					X	X
Trigger processing set-oriented			X	X	X	X		
Conflict resolution: priorities	X	X	X	X	X			X
Conflict resolution: concurrent act.						X		
Conflict resolution: arbitrary act.							X	
Condition check: standard eval.	X	X	X				X	X
Condition check: dedicated eval.				X	X	X		
Transactions: extended	X	X	X	X	X	X	X	
Transactions: separate/nested	X					X	X	
Different coupling modes	X					X	X	

References

Agrawal, R, Cochrane, R and Lindsay, B, 1991, "On Maintaining Priorities in a Production Rule System" In: Camps Lohman, S, ed., *Proceedings 17th VLDB Conference* pp 479–487, Barcelona.

Aiken, A, Widom, J and Hellerstein, JM, 1992, "Behaviour of Database Production Rules: Termination, Confluence, and Observable Determinism" In: *Proceedings ACM SIGMOD Int. Conf. on Management of Data*, San Diego, CA.

Anderson, JA, 1983, *The Architecture of Cognition* Harvard University Press.

Bahr, E, Barachini, F and Mistelberger, H, 1992, "A New Parallelization Method for Production Systems" In: *Proceedings Int. Conf. on Fifth Generation Computer Systems* pp 969–976, Tokyo, Japan.

Bancilhon, F, Delobel, C and Kanellakis, P, 1992, *Building an Object-Oriented Database System: the story of O2* Morgan Kaufmann.

Barthez, M, Kiernan, G, Maindreville, C and Simon, E, 1988, "An Overview of the RDL1 Deductive Database System" *Technical Report N-1133*, INRIA, Rocquencourt, France.

Beeri, C and Milo, T, 1991, "A Model for Active Object Oriented Database" In: Camps Lohman, S, ed., *Proceedings 17th VLDB Conference* pp 337–349, Barcelona.

Blakeley, JA, Coburn, N and Larson, P, 1989, "Updating Derived Relations: Detecting Irrelevant and Autonomously Computable Updates". *ACM Trans. Database Systems.* **14**(3) 369–400, September.

Buchanan, BG and Mitchell, T, 1978, "Model-directed learning by production rules" In: Waterman, DA and Hayes-Roth, F, eds., *Pattern-Directed Inference Systems* Academic Press.

Buneman, OP and Clemons, EK, 1979, "Efficiently Monitoring Relational Databases" *ACM Trans. Database Systems.* **4**(3) 368–382, September.

Cacace, F and Ceri, S, 1990, "Integreating Object-Oriented Data Modelling with a Rule-Based Programming Paradigm" In: Garcia-Molina, H and Jagadish, HV, eds., *Proceedings ACM Conference on the Management of Data (SIGMOD)* pp 225–236, Atlantic City, May.

Caseau, Y, 1991, "An Object-Oriented Deductive Language", *Ann. Mathematics and Artificial Intelligence* **3**.

Ceri, S, Gottlob, G and Tanca, L, 1989, "What You Always Wanted to Know About Datalog (And Never Dared to Ask)" *IEEE Trans. Knowledge and Data Engineering* **1**(1) 146–166, March.

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

Ceri, S and Widom, J, 1990, "Deriving Production Rules for Constraint Maintenance" In: McLeod, ed., *Proceedings 16th VLDB Conference* pp 566–577, Brisbane, August.

Ceri, S and Widom, J, 1992a, "Deriving Incremental Production Rules for Deductive Data" Rapporto interno n. 92-081, Politecnico di Milano.

Ceri, S and Widom, J, 1992b, "Production Rules in Parallel and Distributed Database Environments" In: Yuan, ed., *Proceedings 18th VLDB Conference* pp 339–351, Vancouver, Canada, August.

Chakravarthy, S, 1989, "Rule Management and Evaluation: An Active DBMS Perspective" *SIGMOD REC., Special issue on Rule Management and Processing in Expert Database Systems* **18**(3), September.

Chomsky, AN , 1957, *Syntactic Structures* The Hague: Mouton.

Cooper, R, Fox, J, Farringdon, J and Shallice, T, 1992, "Towards a systematic methodology for cognitive modelling".

Czejdo, B, Eick, Ch and Taylor, M, 1991, "TANGUY: Integrating Database, Rule-based and Object-Oriented Paradigms" In: *2nd International Symposium on Database Systems for Advanced Applications* pp 339–346, Tokyo, April.

Davis, R, 1980, "Meta-rules: reasoning about control" *Artificial Intelligence* **15**(3).

Davis, R, 1982, "Applications of meta-level knowledge to the construction, maintenance and use of large knowledge bases" In: Davis, R and Lenat, DB, eds., *Knowledge-based Systems in Artificial Intelligence* McGraw-Hill.

Davis, R and King, J, 1976, "An overview of production systems" In: Elcock, EW and Michie, D, eds., 1976, *Machine Intelligence vol 8* Wiley, 300–332.

Diaz, O, Paton, N and Gray, P, 1991, "Rule Management in Object Oriented Databases: A Uniform Approach" In: Camps Lohman, S, ed., *Proceedings 17th VLDB Conference* pp 317–326, Barcelona, September.

Erman, LD, Hayes-Roth, VR, Lesser, VR and Reddy, RD, 1980, "The HEARSAY-II speech understanding system: Integrating knowledge to resolve uncertainty" *ACM Computing Surveys* **12**(2).

Forgy, CL, 1982, "Rete: a fast algorithm for the man pattern/many object pattern match problem" *Artificial Intelligence* **19**(1) 17–37.

Forgy, C and McDermott, J, 1976, "OPS, a domain-independent production system language" In: *Proceedings 5th International Joint Conference on Artificial Intelligence* Cambridge, MA, 933–939.

Gardarin, G, de Maindreville, C and Simon, E, 1989, *Extending a Relational DBMS Towards a Rule-Based System: An Approach Using Predicate Transition Nets* pp 131–152, Springer-Verlag.

Gehani, N and Jagadish, HV, 1991, "Ode as an Active Database: Constraints and Triggers" In Camps Lohman, S, ed., *Proceedings 17th VLDB Conference* pp 327–336, Barcelona, September.

Gehani, NH, Jagadish, HV and Shmueli, O, 1992, "Event Specification in an Active Object-Oriented Database" In: *Proceedings ACM SIGMOD Int. Conf. on Management of Data* San Diego, CA, June.

Gehani, NH, Jagadish, HV and Shmueli, O, 1992a, "Composite Event Specification in Active Databases: Model and Implementation" In: *Proceedings 18th Int. Conf. on Very Large Databases* Vancouver, Canada, August.

Hajnal, SJ, Fox, J and Krause, PJ, 1989, "Sceptic User manual" *Technical Report* Imperial Cancer Research Fund.

Hanson, EN, 1989, "An Initial report on the Design of Ariel: A DBMS With an Integrated production Rule System" *ACM SIGMOD Record* **18**(3) September.

Hanson, EN, 1992, "Rule Condition Testing and Action Execution in Ariel" In: *Proceedings ACM SIGMOD Int. Conf. on Management of Data* San Diego, CA, June.

Held, G, 1975, "INGRES: A Relational Data Base System" In: *Proceedings National Computer Conference* Anaheim, CA, June.

Hellerstein, JM, 1991, "Determinism in Partially Ordered Production Systems" *IBM research report.*

Ingres Corporation, *Introducing INGRES* Release 6.4"
Iyengar, SS, Sabharwal, A, Pin, FG and Weisbin, CR, 1992, "Asynchronous production system for control of an autonomous mobile robot in real-time environment" *Applied Artificial Intelligence* **6**(4) 485–510.
Kiernan, G, Maindreville, C and Simon, E, 1987, "A Production Rule Language for Databases extended towards the Support of Complex Domains" Technical Report N-1133 INRIA, Rocquencourt, France.
Kiernan, G, Maindreville, C and Simon, E, 1990, "Making a Deductive Database a Practical Technology: a step forward" In: Garcia-Molina, H and Jagadish, HV eds., *Proceedings ACM Conference on the Management of Data (SIGMOD)* pp 237–246, Atlantic City, May.
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.
Laird, JE, Newell, A and Rosenbloom, PS, 1982, "SOAR: an architecture for general intelligence" *Artificial Intelligence* **33**(1).
Lohman, GM, Lindsay, B, Pirahesh, H and Schiefer, KB, 1991, "Extensions to Starburst: Objects, Types, Functions and Rules" *Communications of the ACM* **34**(10) 94–109, October.
Markov, A, 1954, "Theory of Algorithms" National Academy of Sciences, USSR.
McCarthy, DR and Dayal, U, 1989, "The Architecture Of An Active Data Base Management System" In: *Proceedings ACM SIGMOD Conference on Management of Data* pp 215–224, Portland, OR, June.
Meyfarth, R, 1991, "ACTROB: An Active Robotic Knowledge Base" In: *2nd International Symposium on Database Systems for Advanced Applications* pp 330–338, Tokyo, April.
Newell, A, 1973, "Production systems: models of control structures" In: Chase, WG, ed., *Visual Information processing* Academic Press, 463–526.
Newell, A, 1991, *Unified Theories of Cognition* Harvard University Press.
Newell, A and McDermott, J, 1975, "PSG manual". Department of Computer Science, Carnegie-Mellon University.
Newell, A and Simon, HA, 1972, *Human Problem Solving* Prentice-Hall.
Nii, HP and Feigenbaum, EA, 1978, "Rule based understanding of Signals" In: Waterman, DA and Hayes-Roth, F, eds., *Pattern-Directed Inference Systems* Academic Press.
Post, EL, 1943, "Formal Reductions of the general combinatorial decision problem" *American Journal of Mathematics* **65**: 197–268.
Rosenthal, A, Chakravarthy, S and Blaustein, B, 1989, "Situation Monitoring for Active Databases" In: Wiederhold, ed., *Proceedings 15th VLDB Conference* pp 455–463, Amsterdam, August.
Schreier, U, Pirahesh, H, Agrawal, R and Mohan, C, 1991, "Alert: An Architecture for Transforming a Passive DBMS into an Active DBMS" In: Camps Lohman, S., ed., *Proceedings 17th VLDB Conference* pp 469–477, Barcelona, September.
Simon, E, Kiernan, J and Maindreville, C, 1992, "Implementing High Level Active Rules on top of a Relational DBMS" In: Yuan, ed., *Proceedings 18th VLDB Conference* pp 315–326, Vancouver, Canada, August.
Shortliffe, EH, 1976, *Computer-based Medical Consultations: MYCIN* Elsevier.
Stonebraker, M, Hanson, EN and Potamianos, S, 1988, "The POSTGRES Rule Manager" *IEEE Trans. Software Engineering* **14**(7) 897–907, July.
Stonebraker, M and Jhingran, A, 1990, "On Rules, Procedures, Caching and Views in Data Base Systems" In Garcia-Molina, H and Jagadish, HV, eds., *Proceedings ACM Conference on the Management of Data (SIGMOD)* pp 281–290, Atlantic City, May.
Stonebraker, M, Rowe, LA and Hirohama, M, 1989, "The Implementation of Postgres" Technical report, University of California, Berkeley.
Vaghani, J, Ramamohanarao, K, Kemp, D, Somogyi, Z and Stuckey, P, 1990, "Design overview of the Aditi deductive database system" In: *Proceedings NACLP Workshop on Deductive Databases* Kansas State University.
Vere, SA, 1978, "Inductive learning of relational production systems" In: Waterman, DA and Hayes-Roth, F, eds., *Pattern-Directed Inference Systems* Academic Press.
Vieille, L, Bayer, P, Küchenhoff, V, Lefebvre, A and Manthey, R, 1991, "An Overview of the EKS-V1 System" TR-KB-38, ECRC, August.
Waterman, DA and Hayes-Roth, F, 1978, *Pattern-Directed Inference Systems* Academic Press.
Widom, J, Cochrane, RJ and Lindsay, BG, 1991, "Implementing Set-Oriented Production Rules as an Extension to Starburst" In: Camps Lohman, S, ed., *Proceedings 17th VLDB Conference* pp 275–285, Barcelona, September.
Widom, J and Finkelstein, SJ, 1990, "Set-Oriented Production Rules in Relational Database Systems" *ACM Trans. Database Systems* 259–270.
Zhou, Y and Hsu, M, 1990, "A Theory for Rule Triggering Systems" In: Bancilhon, F, Thanos, C and Tsichritzis, eds., *Proceedings 2nd International Conference on Extending Database Technology (EDBT 90)* pp 407–421, Venice, Italy, March.