THE INTEGRATION OF RULE SYSTEMS AND DATA BASE SYSTEMS

Michael Stonebraker
EECS Department
University of California, Berkeley

Abstract

This paper explores the integration of rule systems into data base management systems. One major theme is a survey of the research activities in this area over the past decade. The focus is on prototype systems that have been completely specified and the implementation issues encountered. The second thrust is to present a research agenda which should be addressed by the research community over the next few years.

1. INTRODUCTION

Historically rule management has been the domain of expert systems, and is included in such expert system shells as OPS5 [FORG81], Prolog [CLOC81], and KEE [INTE85]. As a result, rule systems were contained in stand-alone programs with no interaction with DBMS systems.

Clearly, there are many instances where a knowledge base of rules is associated with a data base of facts. For example, consider a personnel data base with a large number of instances of employees, containing their salary, department, amount of accrued vacation, etc. Every company has a collection of rules that indicate who is allowed to be in the pension plan, how much vacation any employee is entitled to, who can have a key to the executive washroom, etc. This rule base is usually called the company's personnel policy, and it can exist in several different forms. First, it can be written down in a booklet and distributed to all employees. In this case, there is no guarantee that the facts in the data base correspond to the rules in the knowledge base. Hence, the **consistency** of the data and the knowledge base is not guaranteed. A second place where the rules can reside is in an application program which accesses the data base. In this case, at least the rules are in computerized form. However, there is still no guarantee that the rules and the data are consistent. For example, a data base update can occur which does not go through the application program. In this case, the data base can be changed to be inconsistent with the knowledge base without the application being aware of it. A second problem with encoding rules in an application program is that they are often difficult to change, and it is usually difficult to bring the data base into conformity with the new rules. As a result, storing rules in an application program (or an expert system shell) has inherent disadvantages.

To help with these problems, there has been considerable effort expended to **couple** expert system shells to data base systems. The KEE connection [INTE87], Bermuda [IOAN87] and KBMS [CERI86] are examples of this sort of activity. Such interfaces allow the expert system shell to interact with the data base more easily. However, they do not solve the consistency problem inherent in an architecture where the data is in one system and the rules are in another one.

The third place where a knowledge base can reside is inside the DBMS. In this case, the DBMS can make an iron-clad guarantee that the rules and the data are consistent, because a single system is administering both kinds of objects. Moreover, there are cases where much better performance will result from an

This research was sponsored by the Army Research Office under Contract DAAL 03-87-0083.

integrated approach. Because of these potential advantages there has been dramatic effort in integrating rules and data over the past decade. Essentially all of this activity has been performed by the data base research community. Furthermore, the results of this research have already been transferred to commercial DBMSs. For example, both INGRES and Sybase have fairly sophisticated rules systems integrated into their relational DBMS engines.

In this paper, we briefly survey the research efforts that have been made in the past decade in integrating rules and data. In Section 2 we indicate the major ideas that have been explored and then turn in Section 3 to the implementation tactics that can be used. The paper next addresses the major difficulties that remain to be solved. In my opinion, these are issues concerning the semantics of rules, higher level abstractions for rules, and supporting the capability of querying a rule base. In the final three sections of this paper, we discuss each of these topics.

2. CLASSIFICATION OF DBMS RULE SYSTEMS

The focus of research in integrating rules and data has been on supporting so-called **production** systems, i.e. rules of the form:

on event
do action

It is helpful to categorize such rules into four categories, based on the following:

The event can either be an update event or a retrieve event.

The action can either be an update or a retrieve.

The first category is rules which have an update event and an update action. For the traditional EMP table:

EMP (name, age, salary, dept, manager)

an example of this class of rule using the POSTGRES [STON90] rule system is

on replace to EMP.salary where EMP.name = "Joe"
then do replace EMP (salary = new.salary) where EMP.name = "Sam"

This rule instructs the DBMS engine to watch for an event which is an update to Joe's salary. When this event occurs, the engine should perform the corresponding action, which is to propagate Joe's new salary on to Sam.

Of course, another rule could be defined which would propagate Sam's salary on to a third employee, Fred. In this case an update to Joe's salary would trigger the first rule which would update Sam's salary. In turn this update would trigger the second rule, and a **forward chaining** control flow would result. When there are no more rules to activate, the chaining process stops.

Forward chaining rule systems with syntax similar to the notation above are operational in two prototype next-generation DBMSs, Starburst [WIDO90] and POSTGRES [STON90]. Furthermore, similar notation is supported in commercial relational systems including INGRES [INGR90] and Sybase [SYBA90]. Prototype efforts along the same lines have also been reported in ARIEL [HANS89], HiPAC [MCCA89] and RPL [DELC88]. All systems build on the pioneering work of [ESWA76].

A second class of rules are ones of the form:

on update
do retrieve

For example, consider the following POSTGRES rule:

on replace to EMP.salary where EMP.name = "Joe"
do retrieve (new.salary)

This rule acts as an **alerter,** i.e. whenever Joe gets a salary adjustment, the user defining the rule is sent his

new salary. In this way the second user is **alerted** when an event of interest takes place. Alerters have been studies in [BUNE79]. Of the example rules systems noted above only POSTGRES and HiPAC support this construct.

The third class of rules are ones of the form

on retrieve
do retrieve

Consider, for example, the following POSTGRES rule:

on retrieve to EMP.salary where EMP.name = "Sam"
do instead retrieve (EMP.salary) where EMP.name = "Joe"

The semantics of this rule is to look for an event that is a retrieve to Sam's salary. When the event occurs, the action is to be performed **instead** of the event. Consequently, Joe's salary is returned as a result of the request for Sam's salary. In effect, the stored value of Sam's salary, if any, has no consequence because the salary of Joe is substituted instead.

If Joe's salary is derived according to a second rule of the same form, then a retrieve to Sam's salary would awaken the first rule, which in turn would awaken the second one. Therefore a **backward chaining** control flow would result.

Another way of thinking about rules in this third class is that they allow portions on a data base to be **virtual** or **derived** data items. Commercial relational DBMSs support **views,** which are virtual tables, typically using the algorithms in [STON75]. As a result, backward chaining rules generalize the view concept by allowing finer granularity for virtual objects.

Of course, one could use a Prolog-style syntax for rules in this class, e.g:

salary (Sam) = salary (Joe)

and there have been several systems that implement backward chaining syntax of this sort, including NAIL [ULLM85] and LDL [CHIM90]. The only system known to the author that supports both forward and backward chaining is POSTGRES.

The thrust of research in backward chaining systems has been almost exclusively on processing **recursive** rules. For example, consider another column of the traditional EMP table containing the second-level manager of each employee. This column can be supported by the following backward chaining rule:

on retrieve to EMP.mgr-mgr
then do instead retrieve (E.mgr) where E.name = current.mgr

A generalization of this example might be to find all the the indirect managers of each employee in a field, indr-mgr. In this case, the appropriate rule is:

on retrieve to EMP.indr-mgr
then do instead
    retrieve (current.mgr)
    retrieve (E.indr-mgr) where E.name = current.mgr

Here, the first part of the action identifies the direct manager of an employee while the second part finds all the indirect managers of that person. When the same attribute appears in both the event and the action part of a rule, a recursive execution can result.

This rule is perhaps clearer in a Prolog-style syntax:

indirect-mgr (X,Y) = indirect-mgr (X,Z) and manager (Z,Y)

The semantics of this rule is to assert that an employee, X, is the indirect manager of another employee, Y, if there exists a third employee, Z, such that Z is the manager of Y and X is the indirect manager of Z. Since the same clause, indirect-mgr, appears on both the left and the right-hand side of the rule, this again signifies recursive execution will be required to solve for the indirect managers of any particular employee.

The thrust of NAIL and LDL is to efficiently solve queries that require activating recursive rules, such as the query to find all indirect managers of Joe, i.e:

indirect-mgr (X, Joe)

A survey on some of the techniques employed is given in [BANC86], and additional ones are being explored all the time. It is even possible that the techniques being developed for recursive queries will be more generally applicable [MUMI90]. Since recursion is a large topic, this paper will not consider it further.

The fourth class of rules are ones of the form:

on retrieve ...
do update ...

An example of such a rule is:

on retrieve to EMP.salary
then do append to AUDIT
(name = current.name, salary = current.salary, user = user())

Although few systems besides POSTGRES support this class of rules they provide a valuable function, namely support for an **audit trail.** In the above example, every access to any employee's salary results in an audit record being added to the AUDIT table detailing the access.

In the next section we discus the implementation tactics that have been used to support rules of the above four classes.

3. IMPLEMENTATION OF DBMS RULE SYSTEMS

The general problem which must be overcome to support rules in a DBMS is to activate the proper rules at the required time. We begin with techniques appropriate to forward chaining rules, e.g. ones of the form:

on replace to EMP.salary where EMP.name = "Joe"
then do replace EMP (salary = new.salary) where EMP.name = "Sam"

There are three basic techniques which can be applied:

brute force
discrimination networks
marking

Brute force entails maintaining a list of all rules that affect each table in a data base. Then, each individual update is **matched** against the condition part of each rule in the list to determine which must be awakened. Clearly, this is a sequential search that will only provide good performance for a very small number of rules per table. With a larger number of rules, this list must be organized for efficient access. Discrimination networks, such as RETE [FORG81] and TREAT [MIRA87], have been widely used in expert system shells to speed up this search. The last technique is to utilize a **marking** system to speed rule activation. Here, instead of maintaining a list of rules per table or a discrimination network for this list, the system uses record marking. In this case, each rule is processed against the data base and every record satisfying the event qualification is identified. Each such record is **marked** with a flag identifying the rule to be awakened. As a result, each record is marked with zero or more flags indicating the rules to awaken if various events occur to this record. Of course, this potentially requires much more space than a list or discrimination based approach, however, it offers dramatically better performance, because no searching must be done at run time. Moreover, in the case that most rules have a small scope, i.e the event covers only a few records, then the space penalty will not be severe. There are difficult problems with keeping the markings correct as updates are made to the data base; for further information on this issue consult [STON90].

At the current time all forward chaining implementations known to the author support brute force, one system, ARIEL [HANS90], has specified a discrimination network, and one system, POSTGRES

additionally supports marking. It is expected that discrimination networks will become a more popular implementation technique for DBMS rule engines as rule sets become larger.

A fourth implementation technique is popular in backward chaining implementations, namely **query** substituted into the user command to produce a **modified** command. Consider, for example the rule:

on retrieve to EMP.salary where EMP.name = "Sam"
do instead retrieve (EMP.salary) where EMP.name = "Joe"

and the command:

retrieve (EMP.salary) where EMP.name = "Sam"

In this case, the user command can be effectively modified to:

retrieve (E.salary) where E.name = EMP.name and EMP.name = "Sam"

This can be simplified to:

retrieve (E.salary) where E.name = "Sam"

The algorithms to perform query rewrite are detailed in [STON90] and are generalizations of the query modification techniques written down originally for views in [STON75]. All backward chaining DBMS rules systems (POSTGRES, NAIL, LDL) use query rewrite as their implementation tactic. In addition, [STON90] shows how to extend query rewrite to also support forward chaining implementations.

The choice of implementation tactics is obviously an efficiency issue. The performance of some of the above techniques is analyzed using a simulation model in [STON86].

4. RULE SYSTEM SEMANTICS

Rule systems have fundamentally complex semantics. In this section, we indicate 6 different dimensions in which rule systems may make different semantic choices.

4.1. Time of Wake-up

Consider the following rule for the EMP class:

on replace to EMP.salary where EMP.name = "Jones"
then do replace EMP (salary = new.salary) where EMP.name = "Brown"

along with the user command:

replace EMP (salary = 1000) where EMP.dept = "shoe"

The rule can be activated at four different points in time, namely:

1) immediately

The rule can be awakened immediately upon the event occuring, i.e. at the moment when Jones receives a salary adjustment. This is the approach currently taken in POSTGRES.

2) end of command

The rule can be delayed until the end of the command, i.e. until all shoe department employees have been updated. Before the next command in the transaction is processed, the rule can be awakened.

3) end of transaction

Activation of the rule can be delayed until the end of the user transaction. This is the approach taken in STARBURST [WIDO90].

4) after the end of the transaction

Activation of the rule can be delayed until after the end of the transaction. This is one of the options in the HiPAC proposal [MCCA89], and can only be allowed if the rule runs with a different transaction identifier than that of the user command. This is discussed further in the next subsection.

In general, a different result will be observed depending on the time of wake-up. In fact, four different data base states can occur for the four different cases above.

4.2. Transaction Context

Consider the rule:

```
on retrieve to EMP.salary
then do append to AUDIT
    (name = current.name, salary = current.salary, user = user())
```

The above rule can be awakened in the **same** transaction as the user command or it can be activated in a **different** transaction. If it is activated in the same transaction, then a user can retrieve a salary, and then abort his transaction. If so, the auditing action will be undone by the abort, and the user's retrieve will not appear in the audit trail. To get the desired action, the rule must be activated in a different transaction.

In general, different results will be observed depending on which transaction context is chosen.

4.3. Backward Chaining versus Forward Chaining

Consider the example forward chaining rule noted above:

```
on replace to EMP.salary where EMP.name = "Jones"
then do replace EMP (salary = new.salary) where EMP.name = "Brown"
```

On the other hand, a similar effect could be obtained from the analogous backward chaining rule:

```
on retrieve to EMP.salary where EMP.name = "Brown"
then do instead retrieve (EMP.salary) where EMP.name = "Jones"
```

Unfortunately, the above two rules do not have the same semantics. For example, if Jones is deleted from the data base, then Brown's salary will become null in the backward chaining example but will have Brown's last salary in the forward chaining example. As a result there are two different semantics which result from the choice of whether to employ forward or backward chaining.

4.4. Semantics of Backward Chaining

Consider again the rule:

```
on retrieve to EMP.salary where EMP.name = "Brown"
then do instead retrieve (EMP.salary) where EMP.name = "Jones"
```

Table 1 indicates the possible outcomes for the query:

```
retrieve (EMP.name, EMP.salary)
where EMP.name = "Brown"
```

depending on the number of Jones's there are in the data base. Here, the first column shows the semantics that would result if the user query were interpreted to be:

```
retrieve (EMP.name, E.salary)
where EMP.name = "Brown" and E.name = "Jones"
```

	Union Semantics	Error Semantics	Random Semantics
no Jones	0 instances	1 instance with a null salary	1 instance with a null salary
1 Jones	1 instance	1 instance	1 instance
N Joneses	n instances	error	1 instance

Semantics of Brown's salary
Table 1

Such semantics are the natural result of applying **query rewrite** [STON90] to the user's query in order to satisfy the rule.

The second column would result from interpreting the query and the rule as follows:

```
retrieve into TEMP (EMP.name, EMP.salary)
where EMP.name = "Brown"

replace TEMP (salary = EMP.salary)
where EMP.name = "Jones"
and TEMP.name = "Brown"
```

If there is more than one Jones, the above update is **non-functional,** i.e. it replaces Brown's salary with more than one value. Non-functional updates are rejected by many DBMSs including INGRES [STON76]. Any DBMS with this interpretation of the rule and which rejects non-functional updates will generate column 2 of Table 1.

The third column would be generated by a slight modification of the above scenario. The interpretation of the rule is the same as above. However, the non-functional update is not rejected; instead the update occurs and any one of the multiple values is assigned. Any DBMS which defines non-functional updates to have **random** semantics will generate the final column of Table 1.

If there are two or more employees named Jones, then the natural seamntics for the example rule would be to issue an error, thereby choosing to enforce column 2 of Table 1. However, there appear to be examples for which each column provides the correct interpretation. For example, suppose one wanted a field in the EMP table which will record the people one is required to buy a gift for. In this case, one might write the following rule:

```
on retrieve to EMP.buy-gift
then do instead retrieve (EMP.name) where EMP.manager = current.name
```

This rules suggests that each manager must buy gifts for all employees who work for him. The proper interpretation of this rule is union semantics. In general, all three columns of Table 1 are plausible in application-specific circumstances.

4.5. Ordering

Consider the case where two or more rules can be activated at one time. Specifically, consider the following rules:

```
on replace to EMP.salary where EMP.name = "Jones"
then do replace EMP (salary = new.salary) where EMP.name = "Brown"
```

on replace to EMP.salary where EMP.dept = "shoe"
then do replace EMP (salary = 5000) where EMP.name = "Brown"

If Jones is employed in the shoe department, then both rules will determine Brown's new salary. If Brown is not in the shoe department, then his salary will be determined by whichever rule fired last. In this case the following options appear reasonable:

1) One rule is fired, and the other one is ignored

This will be the semantics supported by the **exception** mechanism in the POSTGRES rules system.

2) Both rules fire in random order

3) Both rules fire, but in a pre-determined order

This will be the semantics supported by the **before** and **after** syntax in STARBURST.

If two rules, A and B, can fire, then the possible options are:

activate only A
activate only B
activate B then A
activate A then B
activate both in random order

Obviously, different answers can be observed from all of these possibilities.

4.6. Number of Possibilities

As a result of the above discussion, there are a vast number of possible semantic outcomes for the activation of two rules, A and B. In fact, it would be an interesting exercise to count the number of possible outcomes, given a single user command which activated N rules.

My contention is that this environment is too complex for any data base administrator to understand. Therefore, I feel that it is a crucial research issue to simplify the possible semantics. There are two possible approaches to this task. First, one could reject certain combinations of the above cases as semantically unreasonable. This would be a valuable research contribution. The second approach is to invent higher level notations that hide this layer of complexity, much as a compiler for a programming language hides the ultimate machine language for any given computer. Examples of this second approach are discussed in the next section.

5. HIGHER LEVEL RULE NOTATIONS

One way to alleviate the semantic morass of the previous section is to view the rule specification language discussed in the previous sections as "data base assembly language", i.e. it is a complex low-level notation that only a few compiler writers have to understand. These guru programmers will implement higher level languages, which have much simpler semantics.

In this case the DBMS engine can concentrate on a very efficient implementation of the lower level notation for rules. The code in this engine is then leveraged to perform a wide variety of useful functions. The challenge for DBMS researchers is to invent a multitude of higher level notations. In this section we give two examples of the approach in some detail. Then, we sketch more briefly several additional higher level notations.

5.1. Views and Versions

In this section we discuss the implementation of POSTGRES views and versions. In both cases, required functionality is supported by **compiling** user level syntax into one or more rules for subsequent activation inside POSTGRES.

Views (or virtual classes) are an important DBMS concept because they allow previously implemented classes to be supported even when the schema changes. For example, the view, TOY-EMP, can be defined as follows:

define view TOY-EMP (EMP.all) where EMP.dept = "toy"

This view is compiled into the following POSTGRES rule:

on retrieve to TOY-EMP
then do instead retrieve (EMP.all) where EMP.dept = "toy"

Any query ranging over TOY-EMP will be processes correctly by the POSTGRES rules system.

However, a key problem is supporting updates on views [CODD74]. Current commercial relational systems support only a subset of SQL update commands, namely those which can be unambiguously processed against the underlying base tables. POSTGRES takes a much more general approach. If the application designer specifies a default view, i.e:

define default view LOW-PAY (EMP.OID, EMP.name, EMP.age) where EMP.salary < 5000

then, a collection of default update rules will be compiled for the view. For example, the replace rule for LOW-PAY is:

on replace to LOW-PAY.age
then replace EMP (age = new.age) where EMP.OID = current.OID

These default rules will give the correct view update semantics as long as there are no possible ambiguous updates. In addition, the application designer is free to specify his own update semantics by indicating other update rules. For example, he could define the following replace rule for TOY-EMP

on replace to TOY-EMP.dept
then delete EMP where EMP.name = current.name and new.dept != "toy"

Therefore, default views are available using compilation of view syntax into a collection of rules. Other update semantics can be readily specified by user-written update rules.

A second area where compilation can support desired functionality is that of **versions.** The goal is to create a **hypothetical** version of a class with the following properties:

1) Initially the hypothetical class has all instances of the base class.
2) The hypothetical class can then be freely updated to diverge from the base class.
3) Updates to the hypothetical class do not cause physical modifications to the base class.
4) Updates to the base class are visible in the hypothetical class, unless the instance updated has been deleted or modified in the hypothetical class.

Of course, it is possible to support versions by making a complete copy of the class for the version and then making subsequent updates in the copy. More efficient algorithms which make use of **differential** files are presented in [KATZ82, WOOD83].

In POSTGRES any user can create a version of a class as follows:

create version my-EMP from EMP

This command is supported by creating two **differential** classes for EMP:

EMP-MINUS (deleted-OID)
EMP-PLUS (all-fields-in EMP, replaced-OID)

and installing a collection of rules. EMP-MINUS holds the OID for any instance in EMP which is to be deleted from the version, and is the negative differential. On the other hand, EMP-PLUS holds any new

instances added to the version as well as the new record for any modification to an instance of EMP. In the latter case, the OID of the record replaced in EMP is also recorded.

The retrieve rule installed at the time the version is created is:

```
on retrieve to my-EMP
then do instead
retrieve (EMP-PLUS.all)
retrieve (EMP.all) where EMP.OID NOT-IN {EMP-PLUS.replaced-OID}
        and EMP.OID NOT-IN {EMP-MINUS.deleted-OID}
```

The delete rule for the version is similarly:

```
on delete to my-EMP
then do instead
append to EMP-MINUS (deleted-OID = current.OID) where EMP.OID = current.OID
delete EMP-PLUS where EMP-PLUS.OID = current.OID
```

The interested reader can derive the replace and append rules or consult [ONG90] for a complete explanation. Also, there is a performance comparison in [ONG90] which shows that a rule system implementation of versions has comparable performance to an algorithmic implementation with hard-wired code deep in the DBMS execution engine.

Both of the examples in this section have shown important DBMS function that can be supported with very little code by compiling higher level syntax into a collection of rules.

5.2. Other Rule Compilers

In this section we more briefly sketch several other possible rule compilers. Referential integrity [DATE81] is often stated as a requirement for current relational DBMSs. In most commercial systems, it is hard-wired as special purpose code. However, it is straightforward to build a compiler for referential integrity syntax. For example, for the traditional employee and department relations:

```
EMP (name, age, salary, dept, manager)
DEPT (dname, floor)
```

one often wishes to check on each insert that that the new employee is in a legal department. If not, one wants to add the new department. This is one of the 9 cases of referential integrity stated in [DATE81], and appropriate user-level syntax can be compiled into:

```
on append to EMP
then do append to DEPT (dname = new.dept) where new.dept NOTIN {DEPT.dname}
```

The other 8 cases are equally easily supported. As a result, referential integrity can be supported by code which compiles special purpose syntax into DBMS rules.

A second application is general integrity constraints. One would like syntax such as:

```
integrity constraint on EMP is EMP.salary > 1000
```

This can be compiled into the rules:

```
on replace to EMP.salary
then abort where new.salary <= 1000

on append to EMP
then abort where new.salary <= 1000
```

A complete compiler for this sort of integrity rules is given in [CERI90].

A third example is support for **fragments** of relations [EPST78]. In this case, one wishes a logical relation, EMP, to be the union of two physical tables, EMP1 and EMP2. This is useful when EMP1 and EMP2 are stored in different storage managers or even on different sites in a distributed data base. Often a

distribution criteria is specified to indicate the conditions under which a physical record is allocated to a particular fragment. For example, one might specify:

```
distribute EMP to EMP1 where EMP.salary < 1000
        to EMP2 where EMP.salary >= 1000
```

This distribution criteria can be compiled into the following simple set of retrieve rules:

```
on retrieve to EMP where EMP.salary < 1000
then do instead retrieve (EMP1.all)

on retrieve to EMP where EMP.salary >= 1000
then do instead retrieve (EMP2.all)
```

If name in EMP is a unique key, then the delete rules are:

```
on delete to EMP where EMP.salary < 1000
then do instead delete EMP1 where EMP1.name = current.name

on delete to EMP where EMP.salary >= 1000
then do instead delete EMP2 where EMP2.name = current.name
```

Append and replace rules can be similarly expressed.

One last example will illustrate the power of rules. Consider supporting two copies of a table, EMP, namely EMP1 and EMP2. Suppose there is a table which can (somehow) be guaranteed to be up that indicates the status of each other table, e.g:

```
UP (table-name, status)
```

and consider a user defined function, DIVIDE, which partitions users into two classes and returns a boolean to indicate which class any given user resides in. With these auxiliary definitions, the following retrieve rules for EMP evenly partition reads over the two copies:

```
on retrieve to EMP where DIVIDE(user)
then do instead retrieve (EMP1.all)

on retrieve to EMP where not DIVIDE(user)
then do instead retrieve (EMP2.all)
```

To support retrieves to a table which is currently unavailable, the following extra rules are required.

```
on retrieve to EMP1 where UP.table-name = "EMP1" and UP.status = "down"
then do instead retrieve (EMP2.all)

on retrieve to EMP where UP.table-name = "EMP2" and UP.status = "down"
then do instead retrieve (EMP1.all)
```

Update rules can be similarly defined for copies that will implement a read-one-write-both processing strategy [BERN81]. For example, again assuming that name is a unique key for employees, the replace rule for salaries is:

```
on replace to EMP.salary
do instead replace EMP1 (salary = new.salary) where EMP1.name = current.name
        replace EMP2 (salary = new.salary) where EMP1.name = current.name
```

The above examples have sketched how referential integrity, general integrity rules, fragments and copies can be supported by compilers that target rule language. It appears that several additional DBMS services can be implemented the same way, including protection, extended transaction schemes [DAYA90], materialized views [CERI91, STON90] and partially materialized views [STON90]. It is hoped that the research community will find other higher level notions that can be leveraged from the rules system by the use of compilation.

6. QUERIES ON THE RULE BASE

The last research area which we think urgently needs exploration is support for queries against the knowledge base. In a production system context, this entails being able to **explain** what has happened as a result of a user command. Basically, the user must be able to ascertain what rules are activated during the execution of any command. The following syntax explores at least some of the desired capabilities, and the examples will use the query language, POSTQUEL.

If a user executes a retrieval command, e.g:

retrieve (EMP.salary) where EMP.name = "Bob"

then he will be returned the salary of Bob, regardless of whether it is stored as data or derived by a backward chaining rule. Consider the following variant:

describe (EMP.salary) where EMP.name = "Bob"

This command will return the salary of Bob only if it is stored as data; otherwise, it will return the rule by which the value is derived. In this way a user gets some indication of why the data element has its indicated value.

Often returning the rule which derives Bob's salary is not adequate; the user would like to know the entire derivation path. This can be satisfied by the following:

describe* (EMP.salary) where EMP.name = "Bob"

Here, Bob's salary is to be described as above. If Bob's salary is derived by a backward chaining rule, then the action part of the rule (a retrieve command) is turned into an describe* command and activated. Describe* will thereby trace the entire derivation of a value, and processing will stop only when a data item is encountered.

Often a user would like to execute a command and be made aware of the rules that gets activated on his behalf. For example, he might issue:

replace EMP (salary = 1000) where EMP.name = "Jones"

and he would like to be notified of rule activations. The following syntax helps in this regard:

trace replace EMP (salary = 1000) where EMP.name = "Jones"

The keyword, trace, before any POSTGRES command would cause the corresponding command to be executed and the user to be notified of any rules which are awakened. A similar command:

trace* replace EMP (salary = 1000) where EMP.name = "Jones"

would cause the above notification as well as activate any rules with trace* added to any command in their action part. In this way the entire path would be traced, and not just a single level.

Lastly, suppose a user wishes to explore what would happen if an update were made; however, he does not actually want to run the update. Hence, he wants to know the effect of a trace or trace* without actually performing the update. The following syntax is intended to have the desired effect:

explain replace EMP (salary = 1000) where EMP.name = "Jones"
explain* replace EMP (salary = 1000) where EMP.name = "Jones"

Adding the keyword, explain, to any command would cause the corresponding command to "hypothetically" run so that rule activation can be traced. However, no changes are actually committed to the data base. If rules are always run in the same transaction as the command that awakened them, then explain and explain* can be supported by running the user's command and then aborting his transaction to back everything out. If different transaction contexts are allowed, then it appears that the easiest way to support the explain command is by using the version system noted in the previous section. A version of each relation that the user updates is constructed on the fly and his updates made to the version. Any awakened rule makes its updates to a version of any table which is to be changed. When the chaining process ends, all the versions can be discarded.

This section has indicated some primitive retrieval capabilities that would facilitate querying a knowledge base, either for debugging purposes or just to achieve an "explanation" of the effect of a command. It is hoped that more sophisticated capabilities will be forthcoming in this area.

7. CONCLUSIONS

In a matter of a few years, integrated rule systems have moved from conceptualization, to research prototypes, and finally to commercial systems. It is clear that they will be the major implementation mechanism for a variety of data base services as noted in this paper. In addition, they will support the construction of expert systems that involve data in a data base because they are **uniquely** able to guarantee that the data is **consistent** with the user's rules. No rule system implemented in a front end expert system shell can make this assertion. As a result, I expect a large number of expert systems to be written using this technology, including ones for personnel management (as indicated in the introduction) and purchase order support. Additional ones that control space assignment in company facilities (e.g you must be a vice president to get a private office) and credit risks (e.g. don't approve a disbursement transaction to anyone who owes you more than $1000) seem straightforward.

Such rule systems are fairly mundane in character and will be termed **simple** rule systems. In contrast **hard** expert systems, such as automated physicians and geologists, require large sets of rules, often with many exceptions, and usually require exploration of complex sets of alternatives. Such systems will probably continue to be implemented using front-end expert system shells which offer more sophisticated rule processing, such as control over the rule activation process and activation and deactivation of rule sets.

Therefore, I expect a large number of simple expert systems to be implemented using rules systems integrated in data base systems, and a much smaller number of hard expert systems to use traditional expert system shells. Of course, any hard expert system should use the DBMS-supported rule system whenever possible, since it is closer to the data and more efficient on many rule-oriented tasks. A last problem for the research community is to investigate how hard expert systems can be supported that are partly implemented in front end expert system shells and partly in DBMS rules. In particular, there are **two** rule engines running on two different rule bases, and the two systems must somehow be co-ordinated. It remains to be investigated how to accomplish this task.

REFERENCES

[BANC86] Bancilhon, F. and Ramakrishnan, R., ‘‘An Amateur's Introduction to Recursive Query Processing,’’ Proc. 1986 ACM-SIGMOD Conference on Management of Data, Washington, D.C., May 1986.

[BERN81] Bernstein, P. and Goodman, N., Concurrency Control in Distributed Database Systems," ACM Computing Surveys, June 1981.

[BUNE79] Buneman, P. and Clemons, E., "Efficiently Monitoring Relational Data Bases," ACM-TODS, Sept. 1979.

[CERI86] Ceri, S. et. al., "Interfacing Relational Databases and Prolog Efficiently," Proc 1st International Conference on Expert Database Systems, Charleston, S.C., April 1986.

[CERI90] Ceri, S. and Widom, J., "Deriving Production Rules for Constraint Maintenance," IBM Almaden Research Lab, Report #7348, March 1990.

[CERI91] Ceri, S. and Widom, J., "Deriving Production Rules for Incremental View Maintenance," IBM Almaden Research Lab, Report #7527, May 1991.

[CHIM90] Chimenti, D. et. al., "The LDL System Prototype," IEEE Transactions on Knowledge and data Engineering, March 1990.

[CLOC81] Clocksin, W. and Mellish, C., "Programming in Prolog," Springer-Verlag, Berlin, Germany, 1981.

[CODD74] Codd, E., "Recent Investigations in Relational Data Base Systems," IBM Research, San Jose, Ca., RJ 1385, April 1974.

[DATE81] Date, C., "Referential Integrity," Proc. Seventh International VLDB Conference, Cannes, France, Sept. 1981.

[DAYA90] Dayal, U. et. al., "Organizing Long-Running Activities with Triggers and Transactions," Proc. 1990 ACM-SIGMOD Conference on Management of Data, Atlantic City, N.J., May 1990.

[DELC88] Delcambre, L. and Etheredge, J., "The Relational Production Language: A Production Language for Relational Databases," Proc. 2nd International Conference on Expert Databases, Tysons Corners, Va., April. 1988.

[EPST78] Epstein, R. et. al., "Distributed Query Processing in a Relational Database System," Proc. 1978 ACM-SIGMOD Conference on Management of Data, Austin, Tx., May 1978.

[ESWA76] Eswaren, K., "Specification, Implementation and Interactions of a Rule Subsystem in an Integrated Database System," IBM Research, San Jose, Ca., Research Report RJ1820, August 1976.

[FORG81] Forgy, C., "The OPS5 User's Manual," Carneigie Mellon Univ., Technical Report, 1981.

[HANS89] Hansen,, E., "An Initial Report on the Design of ARIEL," SIGMOD Record, September 1989.

[HANS90] Hansen, E. et. al., A Predicate Matching Algorithm for Database Rules Systems," Proc. 1990 ACM-SIGMOD Conference on Management of Data, Atlantic City, N.J., May 1990.

[INGR90] INGRES Products Division, "INGRES Version 6.3 Reference Manual," Alameda, Ca., Sept. 1990.

[INTE85] Intellicorp, Inc., "KEE Software Development System User's Manual," Intellicorp, Mountain View, Ca., 1985.

[INTE87] Intellicorp, Inc., "KEEconnection: A Bridge Between Databases and Knowledge Bases," Intellicorp, Inc., Mountain View, Ca., 1987.

[IOAN87] Ioannidis, Y. et. al., "Bermuda - An Architectural Perspective on Interfacing PROLOG to a Database Machine," Computer Science Dept., University of Wisconsin, Tech. Report #723, October 1987.

[KATZ82] Katz, R. and Lehman, T., "Storage Structures for Versions and Alternatives," Computer Science Dept., University of Wisconsin, Madison, Wisc., Report #479, July 1982.

[MCCA89] McCarthy, D. and Dayal, U., "The Architecture of an Active, Object-oriented Database System," Proc. 1989 ACM-SIGMOD Conference on Management of Data, Portland, Ore., June 1989.

[MIRA87] Miranker, D., "TREAT: A Better Match Algorithm for AI Production Systems," Proc. AAAI Conference on Artificial Intelligence, August 1987.

[MUMI90] Mumick, I., et.al., et.al., "Magic is Relevant," Proc. 1990 ACM-SIGMOD Conference on Management of Data, Atlantic City, N.J., May 1990.

[ONG90] Ong, L. and Goh, J., "A Unified Framework for Version Modelling Using Production Rules in a Database System," Electronics Research Laboratory, University of California, Berkeley, Ca., Memo #M90/33, April 1990.

[STON75] Stonebraker, M., "Implementation of Integrity Constraints and Views by Query Modification," Proc. 1975 ACM-SIGMOD Conference, San Jose, Ca., May 1975.

[STON76] Stonebraker, M. et. al., "The Design and Implementation of INGRES," ACM Transactions on Database Systems, Sept. 1976.

[STON86] Stonebraker, M. et. al., "An Analysis of Rule Indexing Implementations in Data Base Systems," Proc. 1st International Conference on Expert Data Base Systems, Charleston, S.C., April 1986.

[STON90] Stonebraker, M. et. al., "On Rules, Procedures, Caching and Views in Data Base Systems," Proc. 1990 ACM-SIGMOD Conference on Management of Data, Atlantic City, N. J., May 1990.

[SYBA90] Sybase, Inc., "Sybase V4.0 Reference Manual," Sybase Corp., Emeryville, Ca., June 1990.

[ULLM85] Ullman, J., "Implementation of Logical Query Languages for Databases," ACM-TODS, Sept. 1985.

[WIDO90] Widom, J. and Finkelstein, SA., "Set-oriented Production Rules in Relational Database Systems," Proc. 1990 ACM-SIGMOD Conference on Management of Data, Atlantic City, N.J., May 1990

[WOOD83] Woodfill, J. and Stonebraker, M., "An Implementation of Hypothetical Relations," Proc. 9th Very Large Data Base Conference, Florence, Italy, Sept. 1983.