On Rules and Integrity Constraints in Database Systems

Tok-Wang Ling and Pit-Koon Teo
National University of Singapore

Abstract

Recently, several researchers have proposed incorporating rules into database systems. These rules typically embody several diverse concepts, eg. deductive rules, production rules, and authorisation rules. However, most systems treat these diverse concepts homogeneously. Their roles are often misunderstood because of this lack of distinction.

This paper clarifies the concepts of deductive rules, production rules, authorisation rules and integrity constraints and point out the essential differences among them. A framework for comparison is established. Several problems with current implementations of the rule mechanisms are highlighted. We also highlight the advantages and disadvantages of each of these concepts and the domains for which they are suitable.

0 Introduction

Recently, several researchers have proposed incorporating rules in object-oriented database systems (OODBMSs)[6,25,9]. These rules capture additional semantics (knowledge) for the database. The rule mechanisms employed in these systems often embody several diverse concepts eg. deductive rules, production rules and authorisation rules. The different forms of rules have different functions. However, most systems treat these diverse concepts homogeneously. Their roles are often misunderstood because of this lack of distinction.

Deductive rules[21] provide a mechanism to derive data which are not explicitly stored in the database. This is the concept of virtual or derived data. Deductive rules are more powerful and expressive than relational views. However, support for deductive rules can be problematic with negative information and recursion.

One of the areas where production rules have been successfully used is in the representation of domain knowledge in rule-based expert systems such as MYCIN[23], XCON[3] etc. Triggers and alerters (both are types of production rules) have been used to monitor and react to events occurring in databases[29, 6, 1]. In some OODBMSs eg. POSTGRES [25], triggers are also used to enforce integrity constraints. However, there exists integrity constraints which are very difficult to enforce using triggers.

In a deductive or relational database, integrity constraints are data dependencies which database states are compelled to obey [17]. These constraints represent knowledge which captures the business rules of the application domain. Different ways of representing integrity constraints within a logic formalism have been proposed[2, 5, 17], eg. tuple calculus, closed first order formula, clause etc. A database operation whose completion violates any integrity constraint should be denied. This has given rise to several proposals for integrity constraint checking[13, 16, 19]. An efficient form of incremental integrity constraint checking was proposed in [17].

Authorisation rules [27] are used to control access to sensitive information (eg. salary). Access is **denied** if it leads to a violation of an authorisation rule defined in the database.

This paper has several contributions. Firstly, the differences among deductive rules, production rules, authorisation rules and integrity constraints are highlighted. We establish a framework against which this comparison can be made. Secondly, several problems with current implementations of these notions in OODBMSs are highlighted. Thirdly, the advantages and disadvantages of each of these concepts and the domains for which they are suitable are discussed.

Section 1 describes the framework used for our comparison. Section 2 compares each of these concepts in terms of the framework. Section 3 discusses the use of these concepts in some of the object-oriented database systems eg. IRIS[12], POSTGRES etc. Section 4 points out the advantages and disadvantages of each of these concepts and diswusses the domains for which they are suitable. In section 5, we summarise our comparison. Section 6 concludes the paper.

1. Framework for comparison

We establish a framework against which to compare deductive rules, production rules, authorisation rules and integrity constraints. The basis for comparison in this paper is given in table 1. We discuss each of these factors in the subsequent sections and present a summary of our comparison in Section 5.

Criteria	Description
Underlying Theory	Whether there is some underlying theory, eg logic upon which the concept is based
Possible representations (Syntax)	How the concept is represented syntactically
Semantics	The interpretation or meaning of the concept. Declarative or procedural aspects of the concept are discussed.
Support for negation and recursion	The support of these notions can enhance the expressiveness of these concepts.
Optimisation	Whether optimisation is possible.
Control strategies	Whether the evaluation strategy is forward chaining (bottom-up) or backward chaining (top-down).
Rule activation	how and when the rules are activated.
Impact on the extensional database	whether each activation adds to the extensional database
Possible conflicts	whether conflicts are possible within a set of each type of rules.
Order of activation/ evaluation of rules	whether there is an ordering of each rule within a rule set. The possibility of having a non-deterministic activation of rules is explored.

Table 1.

2 Comparison of Concepts

2.1 Syntax and Underlying Theory

In a deductive database, deductive rules comprises the intensional database and are usually represented by Horn clauses[18]. Horn deductive rules are based on first-order logic. **Logic** can be used as a uniform language for representing database schema, **integrity constraints**, views, data manipulation and as a programming language. For instance, the programming language Prolog[8] is based on a Horn clause subset of logic. Datalog[4] is another deductive rule-based language (in fact a subset of Prolog) designed specifically to interact with databases.

Examples 1 to 3 show the use of Horn clauses in schema representation, views and data manipulation respectively. Example 4 provides some integrity constraints using a logic formalism called IC-formula[17]. In [17], it was shown that an integrity constraint which involves universal quantifiers, existential quantifiers, negative conditions and/or disjunctive conditions can be expressed by an IC-formula easily. IC-formula is more powerful and expressive than some of the other methods of representing integrity constraints, eg. tuple calculus, closed first order formula, clause [2, 5] etc. We refer the reader to [17] for a formal definition of IC-formula.

Example 1: Schema representation. A schema using clauses is given below. This schema will be used in the rest of the paper.

```
        PERSON (Ic#, Name, BirthDate, Address).
        EMPLOYEE (Emp#, SalesAmt, Salary, Deptno).
        MANAGES (Mgrno, Deptno)
        SUPPLIER (S#, Sname, CompanyName).
        PART (P#, Pname, Color).
        SUPPLIES (S#, P#, Qty).
        PARENT (Parent_Name, Child_Name).
```

Example 2: Views. We can use clauses to define views. For example,

```
   HIGH_SAL_EMP(Emp#, Salary):-
             EMPLOYEE(Emp#, SalesAmt, Salary),
                  Salary >= 30000.
```

defines a view of highly paid employees. In Prolog, it is legal to define another EMPLOYEE predicate (view) as:

```
EMPLOYEE(Emp#,IC#,Name,BirthDate,Address,SalesAmt,Salary,Commission):-
   EMPLOYEE(Emp#,SalesAmt,Salary),
        PERSON(IC#, Name, BirthDate, Address),
             Commission = SalesAmt * 0.25.
```

Notice that Commission is a virtual attribute that is derived from the SalesAmt field. Derived data is automatically updated as required by updates to other parts of the schema. For instance, whenever the value of the SalesAmt attribute is changed, the Commission field is automatically changed. The idea of virtual columns has been suggested elsewhere[25].

The EMPLOYEE view models the ISA relationship between the EMPLOYEE predicate and the PERSON predicate. An instance of the EMPLOYEE predicate is an instance of the PERSON predicate. The attributes of the PERSON predicate can be inherited by the EMPLOYEE predicate.

Another view LAZY-SALESMAN, based on the EMPLOYEE view, can be defined as follows:

```
    LAZY-SALESMAN(Emp#,Name,Commission):-
  EMPLOYEE(Emp#,IC#,Name,BirthDate,Address,SalesAmt,Salary,Commission),
                  Salary+Commission < 5000.
```

Example 3: Data Manipulation. In queries, subject to proper authorisation, users may access views and virtual attributes as they would access data from other parts of the schema, eg:

```
    Select Name, Commission
    From   LAZY-SALESMAN.
```

The answering of a query on a deductive database becomes more of a computation rather than simply a retrieval of information[18].

Example 4: Integrity Constraints.

IC1: All parts are either red or blue.

```
    PART(_,_,Color) -> Color= 'red' V Color = 'blue'
```

where '_' represents the anonymous variable[CLOC81].

IC2: No supplier supplies all the parts

```
    SUPPLIER(S#,_,_,_) -> PART(P#,_,_,_), not(SUPPLIES(S#,P#,_))
```

Unfortunately, unlike deductive rules and integrity constraints, production rules are not based on any particular underlying theory. Production rules generally reside in the rule base of a production system [11] and usually take the form:

```
          IF   condition is satisfied
          THEN perform some action(s) or draw some conclusion(s).
```

Note that production rules can make inference in their consequent portion. We call this kind of rules MYCIN-type rules, after the MYCIN expert system that uses this kind of rules. There appears then to be some similarity with deductive rules. However, they differ from deductive rules in at least two areas. Firstly, their execution semantics is different, as we shall see in Section 2.7. Secondly, many rule-based expert systems eg. MYCIN attach certainty factors to their production rules. These factors measure how strongly the conclusions are believed to be true, and allow for probabilistic reasoning. Although certainty factors can also be attached to deductive rules, if we really want to, it is not normally done.

Recently, several database researchers [20, 29, 26] have extended the general form of a production rule to a triple (E, C, A) where E is an event or events being monitored, C is the condition(s) which must be satisfied before the production rule can be activated and A is the action(s) to be performed when the rule is activated. An event can be a database operation (eg. insert, delete etc), a temporal event (eg. an elapsed time), or an application generated event. When A is a sequence of database updates, the production rule functions as a **trigger.** When A is a signal which is despatched to a user or an external program, the production rule functions as an **alerter**.

Example 5: The department in our example may want to automatically reward all employees whose sales amount exceeds $1000 with a $50 increment. This can be done automatically through a trigger defined as follows (using a form similar to that of [WIDO89]):

```
ON UPDATE TO EMPLOYEE.salesamt /* event */
IF EMPLOYEE.salesamt >= 1000   /* condition */
THEN update EMPLOYEE           /* action */
        set Salary = Salary + 100
        where Emp# = (select Emp#
                  from new-updated EMPLOYEE);  .......(T1)
```

Example 6: The department manager will be interested to identify immediately those employees whose SalesAmt falls below a certain threshold. An alerter can be defined as follows:

```
ON UPDATE TO EMPLOYEE.SalesAmt /* event */
IF EMPLOYEE.SalesAmt < 1000    /* condition */
THEN /* alert  manager */      /* action */
```

Note that multiple messages will be sent to the manager if many employees have their SalesAmt updated to below 1000. The 'alert manager' action is implementation-specific and will not be considered.

Some production rules can be considered as instance-oriented rules[29]. These rules are applied once for each data item satisfying the condition part of the rule. For example, the rules in MYCIN are in this class. Such rules are in fact based on propositional calculus.

Authorisation information in relational database systems is typically kept in an access rights table or an access control list. For instance, in SQL/DS, the user's access rights are maintained in the SYSTEM.SYSTABAUTH catalog table. Owners of database resources can grant or revoke access rights to other users which will update the access rights table. Authorisation rules can augment the existing access rights table by handling access conditions that cannot be handled by the usual access rights table.

Example 7: An access rule can state that only a department manager can access his subordinates' data during a specified time period.

Syntactically, such a rule can be coded using a declarative data manipulation language such as SQL. For instance, one way to retrieve the tuples that a manager in the above example can access is as follows:

```
SELECT * FROM EMPLOYEE
        WHERE deptno = (SELECT deptno FROM MANAGES
                        WHERE mgrno = USER)
        and SYSTIME BETWEEN 0800 AND 1730;
```

The keyword USER denotes the person currently logged on. The keyword SYSTIME (which is not part of current SQL, but which we will use for exposition) defines the current time. Therefore, whenever a manager selects from the EMPLOYEE table, the authorisation subsystem can invoke the above query to retrieve the correct, authorised data.

This can be done by, for example, using a query rewriting technique [24,28] that transform a user's query accordingly:

```
SELECT name, address  =>  SELECT name, salary
FROM EMPLOYEE             FROM EMPLOYEE
                 WHERE deptno = (SELECT deptno
                           FROM MANAGES
                                 WHERE mgrno = USER)
                 and SYSTIME BETWEEN 0800 AND 1730;
```

Note that when a new employee's data is inserted into EMPLOYEE, the data is automatically accessible by the manager without any explicit update.

2.2 Semantics

Integrity constraints and authorisation rules have a purely declarative semantics. However, both deductive rules and production rules can be used viewed from a declarative or procedural perspective.

A purely declarative set of deductive rules is best exemplified by a Datalog program. In contrast, a Prolog program has an operational semantics. The rules themselves are declarative, but Prolog has several system predicates such as the cut that makes the program procedural.

Rule-based expert systems are production systems which have been shown to have the power of a Turing machine, ie can be used to model any computable procedure[22]. Viewed from this perspective, production rules have a procedural semantics. However, production rules have a declarative semantics in that their consequents can comprise a series of declarative SQL statements that performs some actions.

2.3 Role in database operations

Each of these rule concepts has a different role in database operations. Deductive rules can be invoked directly from queries using a query language such as SQL. Authorisation rules ensure that users have the authority to access the resource specified in the database operation before the operation can proceed. Integrity constraints are data dependencies that must be true on completion of all database operations. Any operation that violates any constraints is denied. Production rules automatically react to events arising from database operations(eg. select, update, insert etc). In turn, these production rules change the database state which may trigger other production rules, or even the same production rules, causing a cascading activation of rules, which may not terminate.

2.4 Negation

It is possible to have negative information in integrity constraints specified using IC-formula. Authorisation rules can incorporate simple NOT conditions in the authorisation queries, eg. using SQL-NOT conditions[10].

In deductive databases, there is a problem of expressing negative information, and of interpreting rules with negation. For instance, the intuitive definition of a negated literal is to take its complement. Unfortunately, the complement is not well defined and may yield an infinite set [28].

We can make assumptions that will allow negative information to be derived. The open world assumption[21] requires that all facts, both positive and negative, be represented in the extensional database. Unfortunately, the number of negative facts can be overwhelming. Another possibility is to assume a closed world assumption (CWA)[21], under which information which is not declared as an explicit fact and which cannot be inferred from the deductive rules is taken to be false. Unfortunately, the CWA is undecidable. Another less powerful inference rule, the negation as failure rule[7], can be used to infer negative information. This rule states that the proof that P is not provable from a particular logic program is always the exhaustive but unsuccessful search for a proof of P. Implementing negation as failure is easy, but it may not terminate[18].

It is possible to include the NOT primitive into the condition and action portions of production rules, just as it is possible to include conjunction and disjunction operators. However, it may not make much sense to specify a NOT in the event specification of a production rule. The semantics of NOT(event) is not clearly defined.

2.5 Recursion

Consider the following recursive deductive rules:

```
ANCESTOR (X,Y) :- PARENT (X,Y).                  .......(R1)
ANCESTOR (X,Y) :- ANCESTOR (X,Z), PARENT(Z,Y).   .......(R2)
```

Here, the ANCESTOR relation is inferred from the extensional facts given by the PARENT relation and computes the transitive closure of the PARENT predicate. This derived predicate can participate in database queries. For instance, the extended SQL query :

```
select Name, Address
from PERSON
where exists (ANCESTOR ('JIM', PERSON.name));
```

retrieves all persons whose ancestor is JIM.

It is not clear whether the SQL-based view mechanism can be enhanced with the expressive power of deductive rules. Consider the following example from [25]. It gives a view definition for the ANCESTOR relation based on the PARENT(Parent-Name,Child-Name) relation:

```
range of P is PARENT
range of A is ANCESTOR
define view  ANCESTOR (P.all)
define view* ANCESTOR (A.Parent-Name, P.Child-Name)
                  where  A.Child-Name = P.Parent-Name
```

The * is a closure operator that iteratively retrieves from the PARENT relation until the answer set fails to grow. This is equivalent to a least fixpoint computation. Extension to the query optimizer must be made to support this mechanism. Note that the two views are defined with the same name called ANCESTOR. Current DBMSs do not support multiple views with the same name, but this can be implemented.

Current SQL-based view implementations support a simple form of negation using the SQL NOT keyword and allows a view to be defined in terms of other views (a nested view definition). However, we are not aware of any view implementation that supports both negation and recursion. The application of negation and recursion to deductive rules is equally difficult and is the subject of much research.

Recursion is possible in triggers. Consider the trigger T1 given in Example 5. It is entirely possible for a careless programmer to code the following meaningless production rule:

```
On update to EMPLOYEE.salary
if salary > 100
THEN update EMPLOYEE
        set salesamt = salesamt + 1000
        where EMPLOYEE.Emp# = (select Emp#
                                from new-updated EMPLOYEE)
```

It is clear that these two rules recursively trigger each other indefinitely. Therefore, there is always the danger that triggers may not terminate. Unfortunately, because of the lack of a strong theoretical formalism, there is no generally accepted way to handle this situation. Instead, several proposals have been made, eg. use of a time-out mechanism, use of a static rule analysis program[29] etc.

2.6 Impact on Extensional Database.

The activation of deductive rules, integrity constraints checking and authorisation rules do not change the extensional database. Deductive rules infer new facts on the fly. Authorisation rules build up the user access profile, also on the fly. The inferred facts and the user access profile are not physically stored in the database, and do not extend the extensional database. Integrity constraints do not generate any facts at all and hence do not change the database.

Triggers change the extensional database with delete, update, and insert actions. Mycin-type production rules do not change the extensional database, but affect the working memory, which is one part of a production system. Alerters do not change the extensional database and the working memory.

2.7 Order of evaluation/activation of Rules

Integrity constraints clauses and authorisation rules can be evaluated/activated without considering the order of the clauses or rules. However, certain implementations for deductive rules are sensitive to the order of deductive rules, while other implementations are not. This is best exemplified by the Datalog and Prolog. The order of Datalog rules has no impact on query evaluation. In contrast, Prolog uses a depth first search strategy that is dependent on the order of the rules and the literals in each rule. For instance, if the order of the two rules R1 and R2 in section 2.5 is changed in a Prolog program, then the processing of the query ancestor(X,Y) will not terminate. The same query will terminate in a datalog program.

Production rules are also very sensitive to the order in which they are activated. However, the execution semantics of production rules differs from that of deductive rules. When an event happens, a set of production rules may fire in a particular order. When the same event happens again, the same set of production rules may fire in another order and now acts on a different database state. Different actions and results are obtained because of different order of firing of the same set of production rules. In contrast, deductive rules are activated by user queries and always return the same consistent results at different invocations of the same queries.

2.8 Possibility of Conflicts.

It is possible to encounter conflicts with production rules, integrity constraints and authorisation rules. For instance, the following two integrity constraints are in conflict:

All employees must earn a salary of at least $500.
All employees must earn a salary of at least $400.

It is left to an integrity constraint subsystem to ensure that such conflicts do not occur by rejecting the specification of conflicting integrity constraints. Similarly, authorisation rules can possibly be specified that returns two contrasting access profiles, eg. one giving access to a resource, and the other denying access.

Conflicts in production rules can occur in two ways. Firstly, when an event happens, several rules may be candidates to fire. These candidate rules form a conflict set. Unless all these rules are fired simultaneously, a strategy must be available to select the rule to fire from the conflict set. This is called conflict resolution. Secondly, it is possible for the actions of one production rule to interfere with that of another production rule. For instance, the following two simple rules have conflicting actions:

```
On update to EMPLOYEE
IF EMPLOYEE.name = 'John'
THEN update EMPLOYEE
    Set salary = salary + 200
        Where EMPLOYEE.name = 'John';

On update to EMPLOYEE
IF EMPLOYEE.name = 'John'
THEN update EMPLOYEE
    Set salary = salary - 200
```

```
        Where EMPLOYEE.name = 'John';
```

In the above case, John will not get any salary increment, because of the compensating actions of the conflicting rules.

Several proposals have been made to handle the first kind of conflict in production rules, eg. selecting rules arbitrarily, assigning priorities to rules, firing the most recently executed rule, or using the matching rule that has the greatest number of conditions, breaking ties arbitrarily. None of these proposals is based on any specific theory that allows the behaviour of these rules to be analysed. Under these schemes, it is possible to have rules that never fire (eg. schemes using arbitrary selection or priorities), or rules that can fire indefinitely. An example of the latter kind of rules was described in section 2.5.

The firing of the selected rule may produce a database state that can either invalidate the conditions for some of the remaining rules in the conflict set or satisfy the conditions for other rules. The new rules can be added to the conflict set but it is debatable whether the invalidated rules should be allowed to execute or should be removed from the conflict set. Again, there is no underlying theory to handle this situation.

There can be **no conflicting answers** in retrievals from a deductive database. Since no negative information is explicitly stored in a deductive database, only positive information can be a logical consequence of the database. For instance, consider the following trivial rules :

```
good(X) :- person(X), helpful(X).
bad(X) :- person(X), commit_crime(X).
```

Given the following extensional database:

```
person(a).
person(b).
commit_crime(a).
helpful(a).
```

the queries ?good(a) and ?bad(a) both return the answer true. It would seem that there is a conflict in the two answers given. This, however, is not the case because the system has no notion of the semantics of the terms 'good' and 'bad'. Both answers are logical consequences of the above program. Unless an explicit integrity constraint which states that good(X) --> not(bad(X)) is given, no conflict can arise.

2.9 Optimisation and Control Strategy

There is no standard evaluation strategy for query processing. Two common approaches are the top-down and the bottom-up evaluation strategies. The bottom-up approach is basically a forward-chaining (or data driven) process that starts from the extensional database and proceeds to generate further facts from the rules and the facts until the answer set fails to grow. This approach is simpler but generates a lot of useless results because it does not consider the query. The top-down approach is a backward chaining (goal-driven) process that starts from the query and generates further sub-queries which can be recursively processed to enable answers to be obtained from the database. This approach is more efficient because it does not generate unnecessary results, but it is more complex.

Several algorithms have been proposed based on these two approaches, eg. Naive, semi-naive evaluations (bottom-up), and query-subquery evaluation (top-down). See [BANC86] for an overview.

Triggers and alerters can be processed in a forward chaining manner; backward chaining does not make much sense for triggers and alerters. For MYCIN-type production rules, both forward and backward chaining are entirely possible.

Optimisation of a set of production rules is difficult because of the procedural nature of this set. Production rules communicate with each other by placing data in the database. The entire system behaviour changes when new input is placed in the database. Computation in production rules is data-driven and works mainly by side-effects. This makes optimisation difficult.

The checking of whether a database state obeys an integrity constraint or not is called full integrity constraint checking. This is time consuming, but can be improved by using incremental integrity constraint checking[17,2,5]. Authorisation rules are based on queries and therefore employ standard query optimisation techniques[28].

2.10 Further examples

This subsection presents three further examples.

Example 8: Deductive Rules vs Production Rules: Consider the following production rules:

```
On insert to Mother(X,Y)
        insert Parent(X,Y).

On insert to Father(X,Y)
        insert Parent(X,Y).
```

Given the two insertions to the extensional database,

Mother(a,b)
Father(d,c)

the following facts will be generated and physically stored in the

database:

Parent(a,b)
Parent(d,c).

However, when the fact Father(d,c) is deleted, the extensional database will still contain the following facts:

Mother(a,b)
Parent(a,b)
Parent(d,c).

This is an updating anomaly. Consider instead, the deductive rules:

Parent(X,Y) <- Mother(X,Y).
Parent(X,Y) <- Father(X,Y). ....(D1)

The extensional database comprising Mother(a,b) and Father(d,c) will allow facts Parent(a,b) and Parent(d,c) to be inferred. These inferred facts are not stored explicitly. When the fact Father(d,c) is deleted, the only facts derivable from the database are :

Parent(a,b)
Mother(a,b).

The inferred fact Parent(d,c) is removed automatically when the extensional fact Father(d,c) is removed.

There is no updating anomaly.

Example 9: Deductive Rules vs Integrity Constraints: Given the two deductive rules in D1, the addition of an extensional fact Mother(b,d) will generate an inferred fact Parent(b,d). However, consider the case when the two deductive rules are replaced by the integrity constraints:

Mother(X,Y) -> Parent(X,Y) IC3
Father(X,Y) -> Parent(X,Y) IC4

Since no deductive rules are available to generate tuples for the Parent predicate, we consider the following extensional database:

Mother(a,b)
Parent(a,b).

This extensional database satisfies the integrity constraints. However, the addition of the new fact Mother(b,d) will violate the integrity constraint IC3. The operation to insert the new fact Mother(b,d) should be denied.

Example 10: Integrity Constraints vs Production Rules: Production rules have often been cited as a useful mechanism to enforce integrity constraints[25,29,20]. However, there exist integrity constraints that are difficult to enforce using the current implementation of production rules. For instance, consider this constraint:

Suppliers who do not supply all red parts must supply a blue part or a green part with quantity greater than 100.

To enforce this constraint, more than one trigger have to be implemented, as shown in table 2. For example, when a supplier is inserted, the following trigger could be activated (in pseudocode):

```
On insert to SUPPLIER
IF SUPPLIER does not supply all the red parts
and (supplier does not supply any blue part with qty >100
or supplier does not supply any green part with qty >100)
THEN ROLLBACK
```

This is not a trivial trigger to code. For the above constraint, several of such complicated triggers need to be coded. Even if such triggers can be coded, it cannot be verified that these triggers are coded correctly and does what they are supposed to do. Coding triggers is essentially a programmer responsibility as opposed to, say, a transaction which is a DBMS responsibility. As such, a trigger can be improperly coded, or not coded at all. The assumption that a trigger has been coded, when it has not, can lead to undesirable consequences. For example, a trigger to delete all dependents of an employee who just got fired may not have been coded, but a programmer who assumes that such a trigger exists will be left with an unexpected database state.

On the other hand, properly coded production rules can simplify programming. For instance, when an employee is deleted, his dependents can be automatically deleted without explicit programming.

This example also presents several interesting issues regarding the use of incremental integrity constraint checking, as opposed to full integrity constraint checking. Table 2 shows that not all database operations require a check on the integrity constraint.

Database Operation	Relation on	Constraint Checking Needed
Insert	Part	Yes, if part is red
	Supplier	Yes
	Supplies	No
Delete	Part	Yes, if part is blue or green
	Supplier	No
	Supplies	Yes
Update	Part	Yes, if color is updated
	Supplier	No
	Supplies	Yes, if quantity is updated and only if the new quantity is less than 100.

Table 2

3. Rules in Object-oriented Database systems.

Several proposals have been made to incorporate deductive rules in OODBMSs, for example, IRIS, POSTGRES, RDL1[15]. IRIS rules are conjunctive and non-recursive, which is very restrictive. The rule mechanism in POSTGRES has some problems which have been highlighted elsewhere[26], eg. no support for view processing, and problems in controlling rule activation etc. Both the RDL1 and the new POSTGRES rules mechanism are production rules rather than pure deductive rules.

In POSTGRES, alerters and triggers are modelled using the "always" keyword. For example, the following command triggers a delete of all DEPT records for departments with no employees:

```
delete always DEPT where count(EMP.name by DEPT.dname
                                where EMP.dept = DEPT.dname)=0
```

Syntactically, the above rule does not clearly specify the events that can cause it to fire. It is not clear whether a database update will activate the above trigger. For example, inserting an employee should not activate the trigger. It is also not clear how the POSTGRES trigger will interact with transactions. For example, consider a hypothetical transaction that swaps two employees, one of whom is the only employee in the department. If the latter employee gets deleted first in the transaction, the abovementioned trigger will delete the employee's department as well. This is clearly not the transaction's intention.

[6, 20] propose an execution model that specifies how production rules are processed in the context of database transactions. In their model, each rule is an object that can have coupling modes as attributes. These modes decide whether the execution of the rule (if triggered) is immediate, or can be deferred till the end of the triggering transaction, or even spawned as a separate transaction. However, the determination of the best coupling mode for each rule is manual and can be difficult.

Triggers can also be defined to maintain indexes in OODBMSs. For instance, if an attribute of an object class is indexed, then any update to the attribute value can set off a trigger which will update the index[30].

4. Strengths, Weaknesses and Suitable Domains

This section discusses the advantages and disadvantages of each of the rule concepts and suggests suitable domains in which to apply these concepts.

The following are advantages of deductive rules:

a. Facts are generated on the fly and are not stored in the extensional database. Therefore storage is saved.

b. Derived data is automatically updated by changes to other parts of the schema. Derived data is self-maintaining. Therefore there will not be any updating anomalies.

c. Deductive rules are more powerful and expressive compared to relational views. They can be recursively defined and can compute the transitive closure of a relation. Relational query languages are not expressive enough and relational algebra is not computationally complete. For instance, the transitive closure of a relation cannot be evaluated using SQL. SQL Relational views cannot be recursive.

The following are disadvantages of deductive rules:

a. In a query using deductive rules, time must be spent computing these rules.

b. Support for negation and recursion can be problematic.

c. A set of deductive rules may not terminate. Unsound implementation, eg. Prolog, may return the wrong results to queries.

Deductive rules have been applied in domains such as theorem proving. It can be used to capture the relationships among predicates in a deductive database. Inference on these relationships can be made to generate facts on the fly.

The following are advantages of production rules:

a. Triggers simplify programming. For example, to fire an employee, the application just needs to delete the employee. Other employee-related details, eg. dependents, can be deleted by triggers, if necessary.

b. Mycin-type production rules are an easy and relatively natural way to express knowledge. The rules are both accessible and easy to modify.

c. Production rules are powerful enough to model any computable procedure.

The following are disadvantages of production rules:

a. Production rules can fire indefinitely and may go into a loop. They are difficult to program and control.

b. Conflicts are possible in production rules, both during the evaluation of the rules and when they fire. This means that a conflict resolution strategy must be employed.

c. The correctness criteria for production rules are not clearly defined, especially when used with transactions.

d. Triggers change the database states and may produce updating anomalies.

e. The enforcement of a complex integrity constraint using triggers may require the coding of many of such rules. There is no guarantee that the coded triggers are correct.

f. The order of execution of triggers are important and may produce different results on different invocations and different order of invocation of the triggers.

[11] proposed that production rules are suitable for domains in which there are many independent states, and where knowledge can be declaratively expressed. This allows multiple, non-trivial production rules to be written. Production rules are also suitable in domains where the computational process consists of a set of independent actions. Such a process requires only limited communications between actions, similar to the execution semantics for production rules.

The main advantage of integrity constraints is that they define explicitly the business rules and their enforcement ensures the consistency of data in the database. Their major disadvantage is the computational overhead involved in their enforcement. Integrity constraints are used in database schema design, and database operations involving updates.

The main advantage of authorisation rules is that it can handle special access rules that cannot be handled by the existing authorisation schemes in DBMSs (eg. access rights table of SQL/DS, access control lists in VAX/VMS etc). Its main disadvantage is the time involved in building up the access profile of the user. Authorisation rules can augment the existing authorisation schemes in DBMSs.

5. Summary of Comparison of the different rule concepts

The table gives a summary of the comparison of deductive rules, production rules, authorisation rules and integrity constraints.

TYPE FEATURE	Deductive Rule	Production Rule	Integrity Constraint	Authorisat-ion Rule
Adds to extensional database	No	MYCIN-type rule and alerters - No Triggers - Yes	No	No
Possible conflicts	No	Yes	Yes	Yes
Is order important in activating/ evaluating rules	Depends on the implementation. For example, in Prolog, order is important; in Datalog, order is not important. Deductive rules always return the same results through different invocations	Yes; it is possible to get different results with different invocations or different order of invocation of production rules	No	No
Possible loop	Depending on algorithm used to process the rules, a loop is possible	Yes, depending on how the production rules are coded	No	No
Possible representations (Syntax)	Horn clause	IF X THEN Y format (MYCIN-type rule); Event-Condition-Action triple or its variant	Tuple Calculus, Horn clause, IC-Formula, etc.	access-rights tables; Declarativ e SQL statements
Role in Database operations	Can be invoked directly from within a query; used to generate facts to answer queries.	triggered as a result of database state changes that satisfy certain conditions.	semantic condition that must be true on completion of all data-base operations. The database operation that leads to the violation of the integrity constraint	ensures that users have the authority to access the database resource before the operation is allowed to proceed.

TYPE FEATURE	Deductive Rule	Production Rule	Integrity Constraint	Authorisat-ion Rule
			is denied.	
Underlying Theory	Logic	No fundamental theory	Logic	Relational Theory
Semantics	Contrast Prolog's operational semantics with Datalog's declarative semantics	MYCIN-type rules have a declarative semantics. The execution of triggers have a procedural semantics; however, action statements in triggers can be declarative	Declarativ e	Declarativ e
Control Strategy	Backward and/or Forward Chaining	triggers and alerters primarily forward chaining; MYCIN-type rules can be both forward and backward chained.	Not Applicable	Execution of queries to build user profile.
Optimisation	Several algorithms eg. naive, semi-naive evaluations have been proposed to optimise logic programs	Hard to optimise because of its imperative nature;	Incrementa l Integrity Constraint Checking	Query Optimisat-ion techniques
Support for negation	Yes	Possible in the condition and action portion of an E-C-A rule, but a not (E) does not make sense.	Yes	SQL-not
Recursion	Yes	Yes	No	No

6. Conclusion

A framework was established that pointed out the difference among deductive rules, production rules, authorisation rules and integrity constraints. We also pointed out the advantages and disadvantages of each of these concepts and suggest suitable domains where each of these rules can be used.

These rule concepts can be integrated usefully in any database systems. For instance, a user query can be checked first by an authorisation subsystem for possible access violation. If the query is an authorised access, the deductive rules within the query, if any, are parsed and incorporated into the query by the deductive rule subsystem. Then the integrity constraint subsystem checks the final query for any possible integrity constraint violation. If no violation is detected, the query is passed to the query manager for execution. This execution may activate some triggers. When triggers finish, the query commits. This integration needs to be further developed especially in the context of object-oriented database systems.

References

[1] Agrawal & Gehani, ODE (Object Database and Environment): The language and the Data Model, Sigmod Record 1989.

[2] Asirelli, Santis & M. Martelli, Integrity Constraints in logic databases, J. Logic Programming 3, 1989.

[3] Barker & O'Conner, Expert Systems for Configuration at Digital: XCON and Beyond, CACM Vol 32 No 3, Mar 89.

[4] Ceri et al, What you always wanted to know about Datalog (And Never Dared To Ask), IEEE Trans. on Knowledge and Data Engineering, Vol 1 No 1, Mar 89.

[5] Chakravarthy, Minker & Grant, Semantic Query Optimization: Additional constraints and control strategies, Proc 1st International Conference on Expert Database Systems, 1986.

[6] Chakravarthy, Rule Management and Evaluation: An active DBMS perspective, Sigmod Record Vol 18 No 3, Sept 1989.

[7] Clark, Negation as Failure, in Logic and Databases (Galliare & Minker eds), Plenum Press, New York, 1978.

[8] Clocksin & Mellish, Programming in Prolog, Springer Verlag, Berlin, Germany, 1981.

[9] -, Third Generation Database System Manifesto, the Committee for Advanced DBMS Function, Memo No. UCB/ERL M90/28, University of California, Berkeley, Apr 1990.

[10] Date, An Introduction to Database Systems Vol I, Addison Wesley, 4th edition, 1986.

[11] Davis & King, An Overview of Production Systems, Machine Intelligence 8, E. Elcock & D. Mitchie (Eds), Horwood, Chichester, England.

[12] Fishman et al, IRIS: An object oriented database management system, ACM Trans Office Information Syst., Vol 5 No 1, Jan 87.

[13] Henschen, McCune & Nagui, Compiling constraint checking programs from first order formulas, in H. Gallaire et.al. (Eds), Advances in Database Theory, Vol 2 (Plenum Press, New York 1984), Pg. 145 - 169.

[14] Ioannidis & Sellis, Conflict Resolution of rules assigning values to virtual attributes, ACM Sigmod 1989.

[15] Kiernan, de Maindreville & Simon, Making Deductive Database a Practical Technology: a step forward, ACM Sigmod 1990.

[16] Kobayashi, Validating Database Updates, Information Systems, 9(1) (1984).

[17] Ling, Integrity constraint checking in deductive databases using the prolog not-predicate, Data and Knowledge Engineering, Vol 2, 1987.

[18] Lloyd, Foundations of Logic Programming, Springer Verlag, 1984.

[19] J.M. Lloyd, E.A. SSonenberg, R.W. Topor, Integrity Constraint Checking in stratified databases, TR 86/5, Department of Computer Science, University of Melbourne, 1986.

[20] D.R. McCarthy et al, The Architecture of an Active Data Base Management System, Proc ACM Sigmod, May 89.

[21] R. Reiter, On Closed World Data Bases, in Logic and Databases (H. Gallaire and J.Minker, eds) Plenum Press, New York, 1978.

[22] E. Rich, Artificial Intelligence, McGraw Hill 1983.

[23] E.H. Shortliffe, Computer-based Medical Consultations: MYCIN, Elsevier, New York, 1976.

[24] M. StoneBraker, Implementation of Integrity Constraints and Views by query modification, ACM Sigmod Intl Conf. on Management of Data, 1975.

[25] Michael Stonebraker, Eric Hanson, Spyros Potamianos, A Rule Manager for Relational Database systems, in The Postgres Papers, Memo No UCB/ERL M86/85 Jun 87 (Revised), University of California, Berkeley.

[26] Michael Stonebraker et al, The Implementation of POSTGRES, IEEE Transactions on Knowledge and Data Engineering, Vol 2, No 1, Mar 1990.

[27] M.B. Thuraisingham, Mandatory Security in object-oriented database systems, Proc OOPSLA '89, Oct 89.

[28] J. Ullman, Principles of database and Knowledge Based Systems, Vol 1, Computer Science Press, 1988.

[29] J. Widom, S. Finkelstein, A Syntax and Semantics for Set-oriented Production Rules in

Relational Database Systems, ACM Sigmod record, Vol 18, No 3, Sept 1989.

[30] S. Zdonik and D. Maier, Fundamentals of Object-Oriented Databases, in Readings in Object-oriented Database Systems, Morgan Kaufman, San Mateo, Ca., 1990.