THE SIMULATION OF BUSINESS RULES IN ACTIVE DATABASES USING EXPERT SYSTEM APPROACH

Ivan Bruha and Frantisek Franek
Dept. of Computing & Software
McMaster University
1280 Main Street West
Hamilton, Ontario
L8S 4L7 Canada
E-mail: bruha@mcmaster.ca
franek@mcmaster.ca

Vladimir L. Rosicky
Terren Corp.
4711 Yonge Street
Toronto, Ontario
M2N 6KB Canada

E-mail: lanny@terren.com

KEYWORDS

Active Database, Active Rules, Deductive Rules, Business Rules, Rule-based Expert Systems

ABSTRACT

A very active field of research of Database Management Systems (DBMS) is concerned with an augmentation of DBMS by rules. Passive rules (constraints) were the first to be investigated and are now widely accepted. Active rules (triggers) are in the transformation from research to mainstream applications. A more complex form of active rules, business rules, are now the topic of many research efforts. In our paper we present a framework for enhancement of SQL-based DBMS by business rules in mostly declarative form (as opposed to the more usual, but less manageable, procedural form). The techniques used utilize rule-based expert system methodology.

INTRODUCTION OF THE FRAMEWORK

The augmentation of database systems by rules is a very active area of database research (Patton 1999, Ceri and Fraternalli 1997). In general, there are two main categories of rules used with database systems. The first, so-called **deductive rules**, often also referred to as **constraints,** are usually described in a declarative fashion and are a part of the data definition and/or data manipulation language of the database system. Their activity is simple - when a violation of a constraint is detected, the action that caused the violation is terminated - and that is why the emphasis is on the description of the constraint. This kind of deductive rules has been wildly accepted by commercial providers of database systems. Besides the test for consistency (integrity), deductive rules can provide the ability to derive new information from the existing information, e.g. so-called **security views** can be defined using deductive rules. In any case, these rules do not alter the content of the database, therefore they are also referred to as **passive rules** (Ceri and Fraternalli 1997).

The other kind of rules, **active rules**, often also called **triggers**, provide computational actions triggered by an occurrence of a specific event, typically a database operation. The action is a reaction to a given stimulus. Such rules are called active, since their actions may cause a change of the database content, in fact, we can talk of their desirable side effects. It is quite obvious that active rules are usually of the form

event -> action

and that the action part has a procedural character. Active rules themselves can be categorised into two major application groups: the first can be characterised as **repair of violations of integrity rules (constraint violation repair)** and the second as **business rules** (Patton 1999).

The purpose of business rules is to keep the business (application-specific) integrity of the database intact. Many current commercial database systems support triggers in a very simple format:

- a simple language is provided to describe the event (trigger) and when the corresponding action should be activated (*before* or *after* the event)
- the action itself is a so-called *stored procedure* (some systems, e.g. INFORMIX, provide a special language for stored procedures, while others, e.g. DB2, allow any language like C/C++ to be used for stored procedures).

This approach is quite suitable for the constraint violation repair, but much less suitable for explicit design of business rules, the reason being that the procedure triggered by the event may be quite complex.

The goal of the research described in this paper is to provide a formalism and a methodology for design of active rules that would be more conducive to business rules and their intended applications. Since for practical reasons we can neither modify nor augment the functional kernel of a commercial database system directly, we had to come up with an approach how to extend an existing commercial database by a layered architecture that would "sit" on the top of the commercial system and thus simulate business rules and their execution in it. Since the extension is not a part of the database system, it actually simulates the active rules, even though from the user's point of view they seem to be an integral part of the database system. Of course, this framework is not intended as a production system, but as a test bed for our ideas and methods regarding the business rules.

DESCRIPTION OF THE DATABASE SYSTEM EXTENSION

We simulate business rules of a given database system by the means of an expert system augmented with a direct database access through embedded SQL queries. We have a long experience with a rule-based McESE (McMaster Expert System Environment) project (Jaffer 1990,Franek and Bruha 1989a, Franek and Bruha 1989b, Franek and Bruha 1990), and a similar system TESS (Terren Expert System Shell) (Franek at al. 1999) that provide a database access. Thus we opted for a similar SQL-augmentation of McESE (see the following section for details) for the task.

We leave it to the commercial system (triggers) to detect events of our interest, but we simulate the actions by a set of McESE rules. This approach offers to us three very important features:

1. McESE rules provide a high-level declarative description of the action and the procedural aspects are deferred to well-localized atomic predicates, i.e. to a very low-level of description. This contrasts with the purely procedural ways of commercially available active rule formalisms.

2. McESE has two built-in rule resolution strategies based on its approach to uncertainty, the *optimistic strategy* (use the best evaluation) and the *pessimistic strategy* (use the worst evaluation), and together with the stratification requirements and checking, confluence and termination (Ceri and Fraternalli 1997, Aiken at al. 1992) are automatically assured.

3. If a database contains attributes with fuzzy or granulated values, then such values could be easily processed by the augmented McESE system as it is in McESE's nature to allow processing of fuzzy and/or granulated values (Jaffer 1990, Franek and Bruha 1990). This is another enhancement our approach provides, since current commercial systems are not capable of processing of such attributes. The fuzzy approach to constraint repair is not needed, while it is more than desirable for business rules.

SQL-AUGMENTED McESE

The underlying expert system shell, called SQL-augmented McESE, is a rule-based expert system shell with a backward-chaining inference engine (Franek and Bruha 1989a, Franek and Bruha 1989b). The language of McESE-rules is rather general and thus quite expressive. Unlike "pure" rule-based expert systems where the knowledge is represented exclusively in a declarative form, McESE introduces a certain form of hybridization, for it allows to some degree a procedural knowledge representation as well. Before we can explain the previous statement exactly, we have to describe a general form of McESE rules.

Rather than providing a formal definition (e.g. using BNF rules or some other formalism), an example of a rule exhibiting all of possible features will illustrate it sufficiently:

r1: 0.3*P1("abc")[>=.3] & ~P2(x,y) & P3(2,3.4,y)
=f =>
S(y,x)[>.6]

r1 is the ***rule identifier***. The left-hand side of the rule is a conjunction of ***terms***, where a term consists of a ***weight*** (may be omitted, otherwise it must be a numeric literal whose value is between 0 and 1 inclusive), a ***predicate*** with list of ***arguments*** (the list may be empty), the arguments may be ***strings***, ***integers***, ***reals***, or ***typed variables*** that can attain values of one of the types mentioned above. The predicate may be negated (~). The predicate may be followed by a ***threshold directive***.

The threshold directive has the form **[op val]** where **op** is a relational operator (one of =, !=, <, <=,>,>=) and **val** is a numeric literal whose value is between 0 and 1 inclusive. The right-hand side of the rule consists of a single term.

The **arrow** `=f=>` defines which ***certainty value propagation*** (of ***cvp*** for short) ***function*** should be used to propagate the uncertainty evaluation of the left-hand side of the rule to the right-hand side term.

A disjunction of terms is modeled by a set of rules, e.g. ***P*** => ***Q*** and ***R*** => ***Q*** are used instead of ***(P or R)*** => ***Q***. The system allows to specify the rule resolution principle either as the **optimistic strategy** (use the rule that gives the best evaluation) or the **pessimistic strategy** (use the rule that gives the worst evaluation). For example, if we always use cvp function *max* for each rule and the pessimistic rule resolution strategy, we get a typical fuzzy logic expert system. McESE is capable of modeling many other approaches to dealing with uncertainty. (Jaffer 1990, Franek and Bruha 1990).

When we say that a McESE rule fires, first the left-hand side expression is evaluated. The evaluation process consists of evaluation of individual terms. The value of a negated term is a complement in 1, i.e. *val*(~t)=1-*val*(t). The value of a term is obtained in the following manner: first the value of the predicate with concrete arguments (i.e. all variables must be instantiated) is obtained (for more exact description see (Franek and Bruha 1989a, Franek and Bruha 1989b, Franek at al. 1999)). If the term contains a threshold directive then the value of the predicate is further modified, otherwise it is not. The modification is based on whether the value of the predicate satisfies the threshold directive, if so, the value is modified to 1, otherwise reduced to 0. If a weight is specified, the resulting value is just multiplied by the weight.

Once the values of all left-hand side terms are known, the prescribed cvp function assigns to the right-hand side predicate its certainty value (it again may be modified by negation, weight, or threshold directive). During evaluation, when a rule is fired, all its left-hand side terms must be evaluated, which may trigger firing of other rules in the backward-chaining fashion. This process chains backward until it stops at ***atomic predicates*** (these are predicates that never occur on a right-hand side of any rule). They correspond to the knowledge of the "real world" and are processed by corresponding ***atomic predicate procedures***

that, based on the arguments passed to them, return certainty values. From that point on, these basic values are propagated forward as described above until they reach the required predicate.

In order to provide fast processing, McESE rule bases are compiled into complex linked data structures and that is what the inference engine works with. The compilation process also organizes the rules in a systematic way, thus eliminating possible side effects introduced by the order of the rules in the rule base and checking the stratification of the rule-base.

The SQL-augmented McESE rules have built-in relation predicates (ISNULL, ISNOTNULL, =, !=, >, >= ,<, <=) and allow expressions using built-in operators + , - , * , and /. An additional value is included, ***datetime***, whose precise format depends on the database system used. Moreover, an SQL query is allowed at any position where a value is valid. Also an SQL query can stand in the place of an atomic predicate. Thus, the SQL-augmented rule set can include and be predicated on results of SQL queries and have as its action an SQL query resulting in the modification of the database.

CASE STUDY

In this paragraph we shall briefly describe and discuss a case study of automatic inventory block release simulated by SQL-augmented McESE.

In travel industry, a retailer like a travel agency may purchase a block of services/goods from a supplier for a particular day (we refer to it as *inventory block*). If the retailer cannot sell the services/goods (or their portion) by a certain time, the inventory (or its portion) is then released using different strategies. The purpose of the release is to stop selling the services/goods from the block in the usual manner, so in a sense the unsold part or its portion is "returned" to the supplier. The usual strategies include request to the supplier for each additional sale, or complete stop of selling, or a reduction of the allotment which can be sold.

For example, a travel agency may purchase a block of airline tickets for a particular flight and if they are not sold by 30 days prior to the flight, the allotment of tickets that can be sold, is reduced by 50% and thus the travel agency can sell fewer tickets and all additional sales must be done through the airline; 5 days prior to the flight, the whole allotment of tickets that can be sold is reduced to 0 and thus travel agency cannot sell any more tickets on its own and all sales must be done through the airline.

Current applications deal with such problems using regularly scheduled inventory block release programs that search through the database and determine if an inventory block ought to be released and how. Inventory block release is an example of an activity that should be left to the database itself rather than to the application or applications, as there is no need to worry about scheduling and whether the inventory block release program ran and when. Moreover, the release is only important when a new sale is actually happening, and thus a regularly scheduled inventory release may be very well an unnecessary effort. Its business logic is self-contained and requires no interaction with the core application(s). Usually inventory block release strategies are described in a form of rules and their procedural aspects are not too complex. Thus, it is a prime candidate for business rules approach, and that is why we selected it as our case study. In this way, the inventory block release became truly automatic.

Each inventory block for a given supply is described in a table *inventory_block*. For simplicity we omit all attributes that are not relevant for our study and also their data types. Thus the table *inventory_block* has attributes

- **BlockId** (unique block identifier)
- **SupplyId** (unique supply identifier)
- **Date** (date for which the inventory in the block is destined)
- **Status** (there are three possible values: ***FREE*** indicating that any unsold quantity from the inventory block can be freely sold, ***REQST*** indicating that any sale must go through a request to the supplier, and finally ***STOP*** indicating no further sale is possible)
- **Allotment** (the original quantity)
- **Unsold** (unsold portion of the original allotment).

When a booking application tries to sell some quantity from an inventory block for a given supply and a given date, it checks **Status** and **Unsold** in *inventory_block* if the requested supply in requested quantity is available for the given date. Then it must update the table *inventory_block*, in particular it updates **Unsold**.

Thus a "before update trigger" is defined for the field **Unsold** which triggers the SQL-augmented McESE inference engine with a specific rule set for inventory block release for this particular supply and this particular date (the system automatically sets value for predefined constants `%supply_id` and `%today`). Since it is a "before" trigger, the inventory release takes place before the update issued by the booking application. If the result of the inventory release invalidates the requested sale (i.e. **Status** has changed to ***REQST*** or ***STOP*** or **Unsold** has been reduced to a quantity less than the quantity requested), the update request of the booking application is failed indicating to the application that the request could not be granted. In what way the booking application deals with the failed update (usually the transaction is rolled back or retried before rolled back) is entirely up to the application and is not relevant for inventory release. (See Fig.1 below)

This setup presents a complication caused by the fact that the inventory block release triggered by the update of **Unsold** may itself update **Unsold** and thus it needs to bypass the trigger. But this is a technicality and not really relevant to the experimentation.

The inventory block release strategies we employed were rather very simple:

Rule1: 20 days prior **Date**, reduce **Unsold** by 50%
Rule2: 10 days prior **Date** change **Status** to ***REQST*** and reduce **Unsold** by 50%

Rule3: 2 days prior **Date** change **Status** to ***STOP***

The corresponding SQL-augmented McESE rule set:

The constants `%supply_id` and `%today` are predefined and their values are set by McESE upon invocation. User-defined constants `%date` and `%status` are defined within the McESE rule set. The constant `%date` refers to the date of the inventory block being processed (as determined by `%supply_id`) and the constant `%status` refers to its status.

```
DEF %date=(SELECT Date
           FROM inventory_block
           WHERE SupplyId=%supply_id)

DEF %status=(SELECT Status
             FROM inventory_block
             WHERE SupplyId=%supply_id)

R1:%date<=%today+20 & %status=FREE
   & ~R2 & ~R3
                  ==>
   (UPDATE inventory_block
    SET Unsold=Unsold*0.5
    WHERE SupplyId=%supply_id)

R2: %date<=%today+10 & %status=FREE
    & ~R1 & ~R3
                  ==>
     (UPDATE inventory_block
      SET Unsold=Unsold*0.5,
          Status=REQST
      WHERE SupplyId=%supply_id)

R3:%date<=%today+2 & ~R1 & ~R2
                  ==>
     (UPDATE inventory_block
      SET Status=STOP
      WHERE SupplyId=%supply_id)
```

Note that we can control the exclusivity of the rule firing and hence whether the rules are performed in exclusive mode (like in the example above when only one of the three rules will fire). If the cumulative effect is desired (which actually is more common), then a simple modification of the rules (see below) will have the desired effect.

```
R1:%date<=%today+20 & %status=FREE
                  ==>
   (UPDATE inventory_block
    SET Unsold=Unsold*0.5
    WHERE SupplyId=%supply_id)

R2: %date<=%today+10 & %status=FREE
                  ==>
     (UPDATE inventory_block
      SET Unsold=Unsold*0.5,
          Status=REQST
      WHERE SupplyId=%supply_id)

R3:%date<=%today+2
                  ==>
     (UPDATE inventory_block
      SET Status=STOP
      WHERE SupplyId=%supply_id)
```

We have presented here the inventory block release business rules in a much simpler form than actually used in the experimentation. The purpose is to allow the reader to understand the fundamental concepts behind our approach, their strength as well as their weaknesses.

The rules can be combined (which is a common situation in real life) with appropriate report activities, e.g. when **Status** is changed to ***REQST***, the appropriate supplier should be notified about the total of sold and unsold services/goods from the original allotment. It should be clear from the description of the workings of McESE that there is no problem to refer to the appropriate activities in McESE rules and hence describe them in the declarative form within the simulated business rules.

We have not addressed yet the issue of storage, i.e. where and how are McESE rules stored and how they are accessed.

Since a McESE rule set is a kin to a computer program (though in a declarative form), it must be stored in a form readily accessible to humans for editing and modifications. On the other hand a speedy processing requires the rule set be pre-processed (compiled) into a proper data structure.

Thus, we opted for McESE rules in a syntax form very similar to SQL queries that were stored in a table of the database in textual form like any other data. Hence our own GUI query builder with syntax checking could be used to create and edit McESE rule sets. A pre-compiler was used to translate such rules to proper McESE rules before they were compiled and stored in binary form in the same table as BLOBs (binary large objects). Any time a rule set was modified, a trigger invoked the pre-compiler and McESE compiler, so the compiled version was kept automatically up-to-date.

We decided not to describe the SQL-like syntax of McESE rules we used in our research, for it did not seem really relevant to the main topic of our research, the simulation of business rules in an active database, and the verification of the expert system approach.

CONCLUSION

The above described case study of automatic inventory block release indicates, despite the simplicity with which it was presented here, that SQL-augmented expert systems like McESE could be successfully utilized as tools for design of active business rules. The main contribution of this approach is the declarative aspect brought to the design of business rules which facilitates understanding and easier development. The additional contribution can be seen in the processing of uncertainty which is very often a part of business activities and cannot be accommodated by the traditional business rules in active databases.

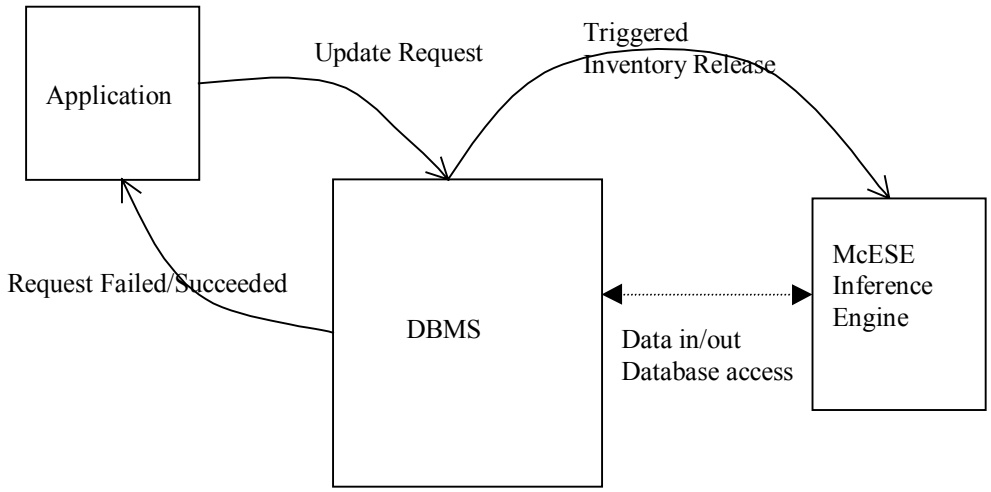

Fig. 1

REFERENCES

Aiken, A.; Widom, J.; and Hellerstein, J.M. 1992. "Behaviour of Database Production Rules: Termination, Confluence and Observable Determinism." *ACM SIGMOD*, vol. 2: 59-68.

Ceri, S. and Fraternalli, P. 1997. *Database Applications with Objects and Rules, The IDEA Methodology*, Addison-Wesley.

Franek, F. and Bruha, I. 1989a. "An environment for extending conventional programming languages to build expert system applications," *Expert Systems Theory & Applications, Proceedings of IASTED International Symposium*, Zurich, Switzerland: 106-109.

Franek, F. and Bruha, I. 1989b. "McESE-McMaster Expert System Environment." In *Computing and Information*, North-Holland: 383-388.

Franek, F. and Bruha, I. 1990. "A way to incorporate Neural Networks into Expert systems." *Artificial Intelligence Application & Neural Networks, Proceedings of IASTED International Symposium*, Zurich, Switzerland: 251-254.

Franek, F.; Rosicky, V.L.; and Bruha, I. 1999. "A hybrid-expert-system based tool for scheduling and decision support." *Proceedings of 13th European Simulation Multiconference ESM99*, Warsaw, Poland.

Jaffer, Z. 1990. "Treatments of Uncertainty and Their Emulation in McESE." *M.Sc. Thesis*, McMaster University.

Paton, N.W., editor. 1999. *Active Rules in Database Systems*, Springer-Verlag, New York.