TBE: A GRAPHICAL INTERFACE FOR WRITING TRIGGER RULES IN ACTIVE DATABASES*

Dongwon Lee, Wenlei Mao, Henry Chiu, Wesley W. Chu
Department of Computer Science
University of California, Los Angeles
Los Angeles, CA 90095, USA
{dongwon,wenlei,hychiu,wwc}@cs.ucla.edu

Abstract One of the obstacles that hinder trigger systems from their wide deployment is the lack of tools that aid users to create trigger rules. Similar to understanding and specifying database queries in SQL3, it is difficult to visualize the meaning of trigger rules. Furthermore, it is even more difficult to write trigger rules using such text-based trigger rule language as SQL3. In this paper, we propose TBE (Trigger-By-Example) to remedy such problems in writing trigger rules by using QBE (Query-By-Example) ideas. TBE is a graphical trigger rule specification language and system to help the users understand and specify active database triggers. TBE retains benefits of QBE while extending features to support triggers. Hence, TBE is a useful tool for novice users to create simple triggers in a visual and intuitive manner. Further, since TBE is designed to hide the details of underlying trigger systems from users, it can be used as a universal trigger interface.

Keywords: Visual Query Interface, Triggers, Active Database

1. INTRODUCTION

Triggers provide a facility to autonomously react to database events by evaluating a data-dependent condition and by executing a reaction whenever the condition is satisfied. Such triggers are regarded as an important database feature and implemented by most major database vendors. Despite their diverse potential usages, one of the obstacles that hinder the triggers from their wide deployment is the lack of tools that aid users to create complex trigger rules in a simple manner. In many

*This research is supported in part by DARPA contract No. N66001-97-C-8601 and SBIRF 30602-99-C-0106.

environments, the correctness of the written trigger rules is very crucial since the semantics encoded in the trigger rules are shared by many applications. Although the majority of the users of triggers are DBAs or savvy end-users, writing *correct* and *complex* trigger rules is still a daunting task, not to mention maintaining written trigger rules.

On the other hand, QBE (Query-By-Example) has been very popular since its introduction decades ago and its variants are currently being used in most modern database products. As it is based on the domain relational calculus, its expressive power is proved to be equivalent to that of SQL that is based on the tuple relational calculus (Codd, 1972). As opposed to SQL, which the user has to conform to the phrase structure strictly, QBE user may enter any expression as an entry insofar as it is syntactically correct. That is, since the entries are bound to the table skeleton, the user can only specify admissible queries (Zloof, 1977). We proposed TBE (Trigger-By-Example) (Lee et al., 99) as a novel graphical interface for writing triggers. Since most trigger rules are complex combinations of SQL statements, by using QBE as a user interface for triggers the user may create only admissible trigger rules. TBE uses QBE in a *declarative* fashion for writing the *procedural* trigger rules (Cochrane et al., 1996). In this paper, we discuss the design and implementation issues of TBE. Further, our design to make TBE a universal trigger rule formation tool that hides much of the peculiarity of the underlying trigger systems is presented.

To facilitate discussion, we shall briefly remind SQL3 triggers and QBE in the following subsections.

1.1. SQL3 TRIGGERS

In SQL3, *triggers*, sometimes called *event-condition-action rules* or *ECA rules*, mainly consist of three parts to describe the event, condition, and action, respectively. Since SQL3 is still evolving at the time of writing this paper, albeit close to its finalization, we base our discussion on the latest ANSI X3H2 SQL3 working draft (Melton, 1999). The following is a definition of SQL3:

Example 1: SQL3 triggers definition.

<**SQL3-trigger**> ::= CREATE TRIGGER <**trigger-name**>
{AFTER | BEFORE} <**trigger-event**> ON <**table-name**>
[REFERENCING <**references**>]
[FOR EACH {ROW | STATEMENT}]
[WHEN <**SQL-statements**>]
<**SQL-procedure-statements**>
<**trigger-event**> ::= INSERT | DELETE | UPDATE [OF <**column-names**>]
<**reference**> ::= OLD [AS] <**old-value-tuple-name**> |

NEW [AS] <**new-value-tuple-name**> |
OLD_TABLE [AS] <**old-value-table-name**> |
NEW_TABLE [AS] <**new-value-table-name**>

1.2. QBE (QUERY-BY-EXAMPLE)

QBE is a query language as well as a visual user interface. In QBE, *programming* is done within two-dimensional skeleton tables. This is accomplished by filling in an example of the answer in the appropriate table spaces (thus the name "by-example"). Another kind of two-dimensional object is the *condition box*, which is used to express one or more desired conditions difficult to express in the skeleton tables. By QBE convention, variable names are lowercase alphabets prefixed with "_", system commands are uppercase alphabets suffixed with ".", and constants are denoted without quote unlike SQL3. Let us see a QBE example. The following schema is used throughout the paper.

Example 2: Define the emp and dept relations with keys underlined. emp.DeptNo and dept.MgrNo are foreign keys referencing to dept.Dno and emp.Eno attributes, respectively.

emp(**<u>Eno</u>**, **Ename**, **DeptNo**, **Sal**)
dept(**<u>Dno</u>**, **Dname**, **MgrNo**)

Then, Example 3 shows two equivalent representations of the query in SQL3 and QBE.

Example 3: Who is being managed by the manager 'Tom'?

```
SELECT E2.Ename
FROM emp E1, emp E2, dept D
WHERE E1.Ename = 'Tom' AND E1.Eno = D.MgrNo AND E2.DeptNo = D.Dno
```

emp	Eno	Ename	DeptNo	Sal
	_e	Tom		
		P.	_d	

dept	Dno	Dname	MgrNo
	_d		_e

The rest of this paper is organized as follows. Section 2 gives a brief introduction to TBE. Section 3 is a simulation of a user session with TBE. The design and implementation of TBE is discussed in Section 4. Section 5 presents the design of some extensions that we are planning for the TBE. Related work and concluding remarks are given in Sections 6 and 7, respectively.

2. TBE (TRIGGER-BY-EXAMPLE)

Triggers in SQL3 are procedural in nature. As shown in Example 1, trigger actions can be arbitrary SQL procedural statements. Also, the order among action statements needs to be obeyed faithfully to preserve the correct semantics. On the contrary, since QBE is a declarative query language, the order is immaterial. Further, QBE is specifically designed as a tool for only: 1) data retrieval queries (i.e., SELECT), 2) data modification queries (i.e., INSERT, DELETE, UPDATE), and 3) schema definition and manipulation queries. Thus, our goal is to develop a tool that can represent the *procedural* SQL3 triggers in its entirety while retaining the *declarative* nature of QBE as much as possible.

2.1. TBE MODEL

SQL3 triggers use the ECA (Event, Condition and Action) model. Therefore, triggers are represented by three independent E, C, and A parts. In TBE, each E, C, and A part maps to the corresponding skeleton tables and condition boxes separately. To differentiate among these three parts, each skeleton table name is prefixed with its corresponding flags, `E.`, `C.`, or `A.`. The condition box in QBE is extended similarly. For instance, a trigger condition statement can be specified in the `C.` prefixed skeleton table and/or condition box.

C.emp	Eno	Ename	DeptNo	Sal

C.conditions

SQL3 triggers allow only INSERT, DELETE, and UPDATE as legal event types. QBE uses `I.`, `D.`, and `U.` to describe the corresponding data manipulations. TBE thus uses these constructs to describe the trigger event types. Since INSERT and DELETE always affect the whole tuple, not individual columns, `I.` and `D.` must be filled in the leftmost column of the skeleton table. Since UPDATE event can affect individual columns, `U.` must be filled in the corresponding columns. Otherwise, `U.` is filled in the leftmost column to represent that UPDATE event is monitored on all columns. Consider the following example.

Example 4: Skeleton tables (1) and (2) depict INSERT and DELETE events on the `dept` table, respectively. (3) depicts UPDATE event of columns `Dname` and `MgrNo`. Thus, changes occurring on other columns do not fire the trigger. (4) depicts UPDATE event of any columns on the `dept` table.

(1)

E.dept	Dno	Dname	MgrNo
I.			

(2)

E.dept	Dno	Dname	MgrNo
D.			

(3)

E.dept	Dno	Dname	MgrNo
		U.	U.

(4)

E.dept	Dno	Dname	MgrNo
U.			

Note also that since SQL3 triggers definition requires that one trigger rule monitors only one event, there cannot be more than one row having an `I.`, `D.`, or `U.` flag. Therefore, the same trigger action for different events (e.g., "abort when either INSERT or DELETE occurs") needs to be expressed as separate trigger rules in SQL3 triggers.

2.2. TRIGGERS ACTIVATION TIME AND GRANULARITY

The SQL3 triggers have notions of *event activation time* and *granularity*. Event activation time specifies whether the trigger is executed before or after its event. Granularity defines how many times the trigger is executed for a particular event.

1 The activation time can have two modes, *before* and *after*. The *before* mode triggers execute before their events and are useful for conditioning of the input data. The *after* mode triggers execute after their events and are typically used to embed application logic (Cochrane et al., 1996). In TBE, two corresponding constructs (`BFR.` and `AFT.`) are introduced to denote these modes. The appended "." denotes that these are built-in system commands by QBE convention.

2 The granularity of a trigger can be specified as either a *for each row* or *for each statement*, referred to as *row-level* and *statement-level* triggers, respectively. The row-level triggers are executed once for each modification to tuple whereas the statement-level triggers are executed once for an event regardless of the number of tuples affected. In TBE notation, `R.` and `S.` are used to denote the row-level and statement-level triggers, respectively.

Users express trigger activation time and granularity at the leftmost column of the event skeleton tables using the introduced constructs.

2.3. TRANSITION VALUES

When an event occurs and values change, trigger rules often need to refer to the *before* and *after* values of certain attributes. These values are referred to as the *transition values*. In SQL3, these transition values can be accessed by either transition variables (i.e., `OLD`, `NEW`) for row-level triggers or tables (i.e., `OLD_TABLE`, `NEW_TABLE`) for statement-level triggers. Furthermore, in SQL3, the INSERT event trigger can only use `NEW` or `NEW_TABLE` while the DELETE event trigger can only use `OLD` or `OLD_TABLE` to access transition values. However, the UPDATE event trigger can use both transition variables or tables. In TBE, a couple of special built-in functions (i.e., `OLD_TABLE()` and `NEW_TABLE()` for statement-level, `OLD()` and `NEW()` for row-level) are introduced. The

`OLD_TABLE()` and `NEW_TABLE()` functions return a set of tuples with values before and after the changes, respectively. Similarly the `OLD()` and `NEW()` functions return a single tuple with values before and after the change, respectively. Therefore, applying aggregate functions such as `CNT.` or `SUM.` to `OLD()` or `NEW()` is meaningless (i.e., `CNT.NEW(_s)` is always 1 and `SUM.OLD(_s)` is always same as `_s`). Using these built-in functions, for instance, an event "every time more than 10 new employees are inserted" can be represented as follows:

E.emp	Eno	Ename	DeptNo	Sal
AFT.I.S.		_n		

E.conditions
CNT.ALL.NEW_TABLE(_n) > 10

When arbitrary SQL procedural statements (i.e., IF, CASE, assignment statements, etc.) are written in the action part of the trigger rules, it is not straightforward to represent them in TBE due to their procedural nature. Because their expressive power is beyond what the declarative QBE (thus TBE described so far) can achieve, we instead provide a special kind of box, called a *statement box*, similar to the condition box. The user can write arbitrary SQL procedural statements delimited by ";" in the statement box. Since the statement box is only allowed for the action part of the triggers, the prefix `A.` is always prepended. An example is:

A.statements
IF (X > 10) ROLLBACK;

2.4. TBE **EXAMPLES**

Let us wrap up this section with two illustrating examples. These are typical trigger rules to maintain database integrity constraints.

Example 5: When a manager is deleted, all employees in his or her department are deleted too.

CREATE TRIGGER **ManagerDelRule** AFTER DELETE ON **emp**
FOR EACH ROW
DELETE FROM **emp E** WHERE **E.DeptNo** IN
(SELECT **D.Dno** FROM **dept D** WHERE **D.MgrNo** = OLD.**Eno**)

E.emp	Eno	Ename	DeptNo	Sal
AFT.D.R.	_e			

A.dept	Dno	Dname	MgrNo
	_d		_e

A.emp	Eno	Ename	DeptNo	Sal
D.			_d	

In this example, the `WHEN` clause is missing on purpose. That is, the trigger rule does not check if the deleted employee is in fact a manager or not because the rule deletes only the employee whose manager is just deleted. Note how the _e variable is used to join the emp and dept tables to find the department whose manager is just deleted. The same query could have been written with a condition test in a more explicit manner as follows:

E.emp	Eno	Ename	DeptNo	Sal
AFT.D.R.	_e			

C.dept	Dno	Dname	MgrNo
	_d		_m

C.conditions
OLD(_e) = _m

A.emp	Eno	Ename	DeptNo	Sal
D.			_d	

Example 6: When employees are inserted to the emp table, abort the transaction if there is one violating the foreign key constraint.

```
CREATE TRIGGER AbortEmp AFTER INSERT ON emp
FOR EACH STATEMENT
WHEN EXISTS (SELECT * FROM NEW_TABLE E WHERE NOT EXISTS
                  (SELECT * FROM dept D WHERE D.Dno = E.DeptNo))
  ROLLBACK
```

E.emp	Eno	Ename	DeptNo	Sal
AFT.I.S.			_d	

C.dept	Dno	Dname	MgrNo
¬	_d		

A.statements
ROLLBACK

In this example, if the granularity were `R.` instead of `S.`, then TBE would generate slightly different SQL3 trigger rule as shown below. That is, a row-level trigger rule generated from the same TBE representation would have been:

```
CREATE TRIGGER AbortEmp AFTER INSERT ON emp
FOR EACH ROW
WHEN NOT EXISTS (SELECT * FROM dept D WHERE D.Dno = NEW.DeptNo)
  ROLLBACK
```

Please refer to (Lee et al., 99) for detailed discussion and more examples of TBE.

3. A TBE SESSION EXAMPLE

To give a flavor of TBE, we describe a sample session in this section. Consider the following example.

Example 7: When an employee's salary is changed more than twice within the same year (a variable CURRENT_YEAR contains the current year value), record new values of `Eno` and `Sal` into the `log(Eno, Sal)` table. Assume that there is another table `sal-change(Eno, Year, Cnt)` that keeps track of the employee's salary changes.

Without TBE, a human expert would have written the following trigger rule:

```
CREATE TRIGGER TwiceSalaryRule
```

```
AFTER UPDATE OF Sal ON emp
FOR EACH ROW
WHEN EXISTS (SELECT * FROM sal-change WHERE
      Eno = NEW.Eno AND Year = CURRENT_YEAR AND Cnt >= 2)
BEGIN ATOMIC
      UPDATE sal-change SET Cnt = Cnt + 1
          WHERE Eno = NEW.Eno AND Year = CURRENT_YEAR;
      INSERT INTO log VALUES(NEW.Eno, NEW.Sal);
END
```

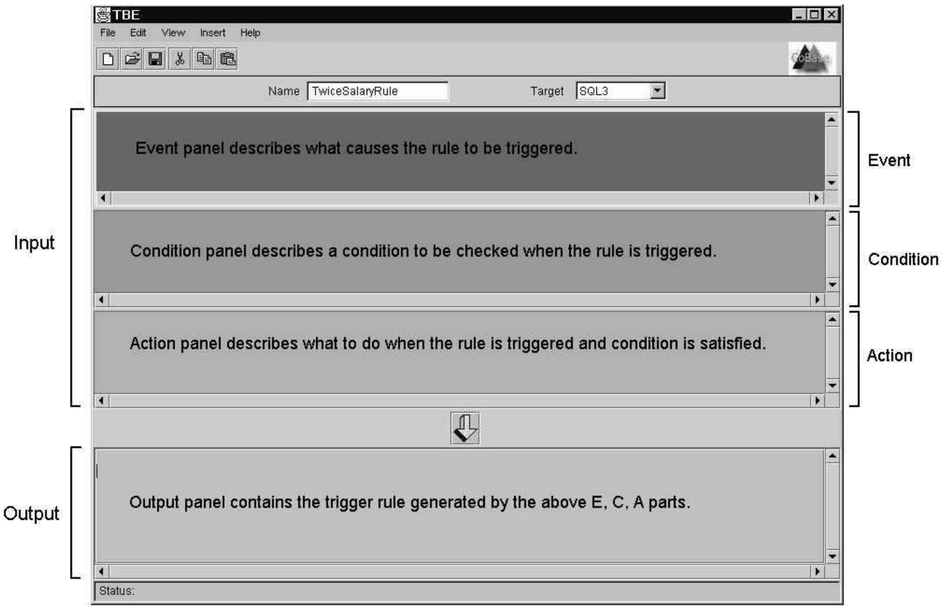

Figure 1 Initial screen.

Initially, TBE looks like Figure 1. Descriptions on the panel are only added for explanation purposes. The main screen consists of two sections – one for input and the other for output. The input section is where the user creates trigger rules by a QBE mechanism and the output section is where the interface generates trigger rules in the target trigger syntax (default is SQL3). Further, the input section consists of three panels for event, condition, and action, respectively. The user first chooses the target system. Then, TBE adjusts its behavior according to the selected target system specifics. Current implementation supports only SQL3 triggers.

At its start-up time, TBE first loads schema information and keeps table, attribute, and type related information. This information is used to guide users to write only admissible trigger rules. For instance, when the user tries to insert an empty skeleton table at one of the three panels, TBE shows all the available table names to aid in the user's selection. After the user picks the table, an empty table appears in the currently active panel.

In our example, the user creates the trigger *event*. From the query description, the user knows that the activation time and the granularity of the triggers are "after" and "for each row", respectively. Furthermore, the `Sal` attribute needs to be monitored for the "update" event (Figure 2). All these commands are provided by TBE and can be chosen from the pop-up menu.

E.emp	Eno	Ename	DeptNo	Sal
AFT.R.				U.

Figure 2 Event construction.

Next, the user constructs the trigger *condition* – "salary is increased more than twice within the same year". To do this, the user can use the fact that "when an employee's salary is updated, if the `Cnt` attribute of the `sal-change` of the same person has value greater than or equal to 2 within the same year, then his update event satisfies the condition". Since the `emp` table needs to be joined with the `sal-change` table to find the candidate employees, the user put variable `_n` in the key attribute (i.e., `Eno`) of the `emp` table. (Figure 3).

E.emp	Eno	Ename	DeptNo	Sal
AFT.R.	_n			U.

Figure 3 A variable inserted at key attribute.

In the `sal-change` table, to specify the same year, CURRENT_YEAR is inserted at `Year` attribute. In addition, to refer to the `Cnt` value later, a new variable `_c` is inserted. Finally, the join condition between `emp` and `sal-change` tables is expressed by entering the variable `_n` in the `Eno` attribute of the `sal-change` table (i.e., equi-join). After constructing "changed more than twice" phrase using the special *condition box*, The resulting TBE is shown in Figure 4.

To facilitate user rule specification, TBE provides the user with all the valid context-sensitive options available for the user to select. For instance, when the user right-clicks after positioning the cursor in the `Eno` attribute, a pop-up menu appears (Figure 5).

Now, the user constructs the trigger *action*. Two actions are required according to the query description: 1) system maintains `Cnt` value in

E.emp	Eno	Ename	DeptNo	Sal
AFT.R.	_n			U.

C.sal-change	Eno	Year	Cnt
	_n	CURRENT_YEAR	_c

C.conditions
_c >= 2

Figure 4 Condition construction.

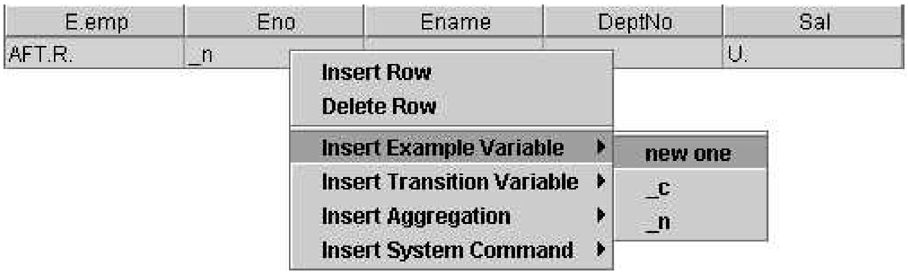

Figure 5 Pop-up menu.

the `sal-change`, and 2) system logs the information of the employee whose salary has been changed more than twice within the same year. Since two actions operate on different tables, the user creates two empty skeleton tables at the event panel. Then, using the variable `_n` defined in the `emp` table, the user increases the `Cnt` value by one (Figure 6).

A.sal-change	Eno	Year	Cnt
U.			_c + 1
	_n	CURRENT_YEAR	_c

Figure 6 Action construction for the `sal-change` table.

Second, the user needs to insert his employee number and his new salary into the `log` table. The user enters another variable in the `Sal` attribute of the `emp` table to refer to the employee's salary value. Furthermore, to retrieve a *new* salary value after an update, the user uses the NEW() function explicitly (Figure 7).

Finally, after the user clicks the down-arrow button to generate the SQL3 trigger rule, the corresponding rule in SQL3 triggers syntax is generated at the output section. Figure 8 shows the final screen after rule generation.

E.emp	Eno	Ename	DeptNo	Sal
AFT.R.	_n			U._s

A.log	Eno	Sal
I.	_n	NEW(_s)

Figure 7 Action construction for the `log` table.

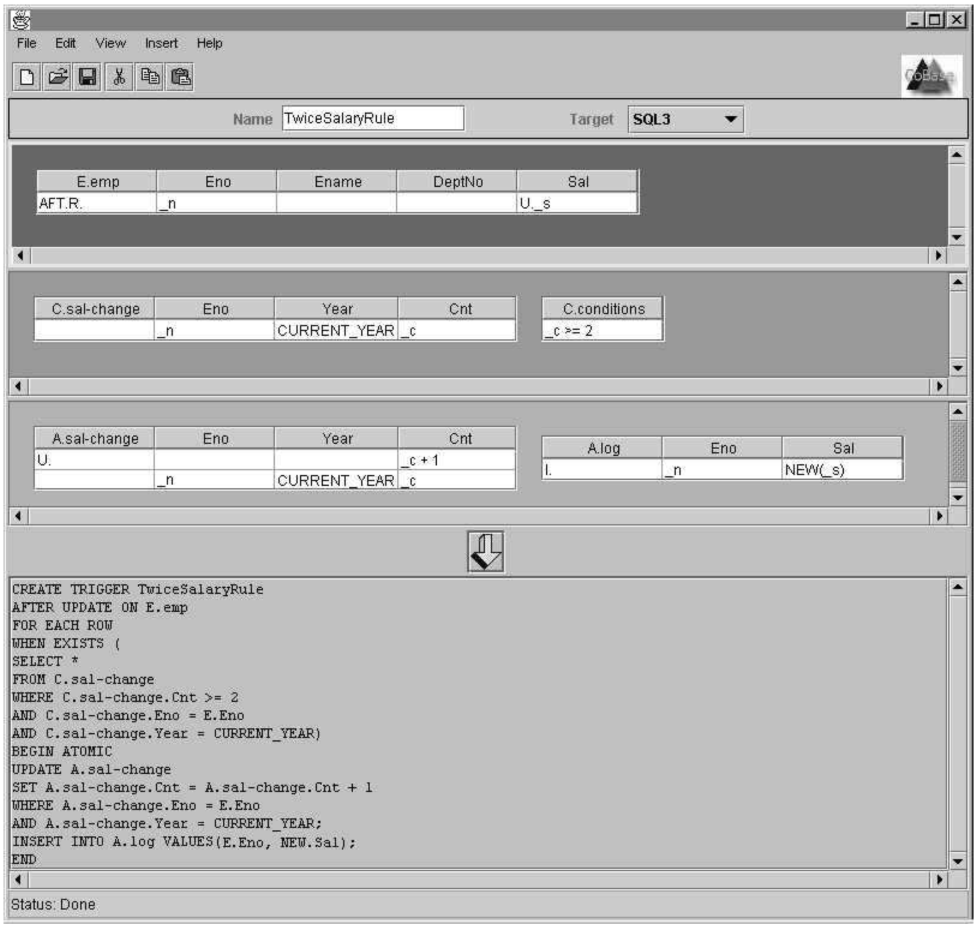

Figure 8 Final screen.

4. DESIGN AND IMPLEMENTATION ISSUES

In this section, we discuss some of the interesting aspects of the TBE implementation. A preliminary version of TBE prototype is being implemented in Java using jdk 1.2.1 and swing 1.1. The main issues that we encountered in designing and implementing TBE are:

- How to represent TBE internally?
- How to implement the translation algorithm?

4.1. INTERNAL REPRESENTATION

Each of the three panels in the GUI (event, condition, and action) holds a vector of tables as created by the user. Before passing the vectors to the translation module, the GUI processes sets (i.e., "[ ]" notation in QBE), removing bracketed entries and replacing them with constants and simple example elements. The modified tables are then used to create internal representations of the tables for the translation module (called TBETables). It contains the column header and a vector of non empty fields. Other useful information such as the fields row and column are stored as well.

The whole session of TBE can be stored on disk using Java's serialization feature.Therefore, current implementation uses the TBETable as an in-memory representation while the serialized object as an on-disk representation of TBE.

For each clause and various checks in the translation algorithm, a linear iteration through the TBETables is required. That is, every time a scan costs $O(N * \bar{M})$, where N is the total number of rows in all TBETables and $\bar{M}$ is the average number of non-empty fields in the rows. Since the size of trigger rule is relatively small, this is not a serious performance problem. One might minimize the constant factor by performing multiple tasks through iterations, but this comes as a cost to modularity.

4.2. TRANSLATION ALGORITHM

Our algorithm is an extension of the algorithm by (McLeod, 1976), which translates from QBE to SQL. Its input is a list of skeleton tables and the condition boxes while its output is a SQL query string. Let us denote the McLeod's algorithm as qbe2sql(<input>) and ours as tbe2triggers.

4.2.1 The qbe2sql Algorithm. We have implemented basic features of the qbe2sql algorithm in (McLeod, 1976), except queries having the GROUP-BY construct. The algorithm first determines the type of query statement. The basic cases involve operators, such as SELECT, UPDATE, INSERT, and DELETE. Special cases use UNION, EXCEPT, and INTERSECT where the statements are processed recursively. General steps of the translation implemented in TBE are as follows:

1 Duplicate tables are renamed. (e.g., “FROM supply, supply” is converted into “FROM supply S1, supply S2”)

2 SELECT clause (or other type) is printed by searching through TBETables' fields for projection (i.e., `P.` command). Then, FROM clause is printed from TBETable table names.

3 Example variables are extracted from TBETables by searching for tokens starting with "_". Variables with same names indicate table joins; table names and corresponding column names of the variables are stored.

4 Process conditions; variables are matched with previously extracted variables and replaced with corresponding table and column names. (e.g., a variable `_n` at column `Eno` of the table `emp` is replaced to `emp.Eno`). Constants are handled accordingly as well.

4.2.2 The tbe2triggers Algorithm.

Let us assume that *_var* is an example variable filled in some column of the skeleton table. `colname`(*_var*) is a function to return the column name given the variable name *_var*. Skeleton tables and condition or statement boxes are collectively called as *entries*.

1 *Preprocessing*: This step does two tasks: 1) reducing TBE query to an equivalent, but simpler form by moving the condition box entries to the skeleton tables, and 2) partitioning the TBE query into distinct groups when multiple trigger rules are written together. This can be done by comparing variables filled in the skeleton tables and collecting those entries with the same variables being used in the same group. Then, apply the following steps 2, 3, and 4 to each distinct group repeatedly to generate separate trigger rules.

2 *Build event clause*: Input all the `E.` prefixed entries. The “`CREATE TRIGGER` <trigger-name>” clause is generated by the trigger name <trigger-name> filled in the name box. By checking the constructs (e.g., `AFT.`, `R.`), the system can determine the activation time and granularity of the triggers. The event type can also be detected by constructs (e.g., `I.`, `D.`, `U.`). If `U.` is found in the individual columns, then the “`AFTER UPDATE OF` <column-names>” clause is generated by enumerating all column names in an arbitrary order. Then,

(a) Convert all variables $\_var_i$ used with `I.` event into `NEW(`$\_var_i$`)` (if row-level) or `NEW_TABLE(`$\_var_i$`)` (if statement-level) accordingly.

(b) Convert all variables $\_var_i$ used with `D.` event into `OLD(`$\_var_i$`)` (if row-level) or `OLD_TABLE(`$\_var_i$`)` (if statement-level) accordingly.

(c) If there is a condition box or a column having comparison operators (e.g., $<$, $\geq$) or aggregation operators (e.g., `AVG.`, `SUM.`), gather all the related entries and pass them over to step 3.

3 *Build condition clause*: Input all the `C.` prefixed entries as well as the `E.` prefixed entries passed from the previous step.

(a) Convert all built-in functions for transition values and aggregate operators into SQL3 format. For instance, `OLD(`$\_var$`)` and `SUM.`$\_var$ are converted into `OLD.name` and `SUM(name)` respectively, where `name` = `colname(`$\_var$`)`.

(b) Fill `P.` command in the table name column (i.e., leftmost one) of all the `C.` prefixed entries unless they already contain `P.` commands. This will result in creating "`SELECT` $table_1$`.*, ..., ` $table_n$`.* FROM` $table_1$`, ..., ` $table_n$" clause.

(c) Gather all entries into <input> list and call qbe2sql(<input>) algorithm. Let the returned SQL string as <condition-statement>. For row-level triggers, create "`WHEN EXISTS` (<condition-statement>)" clause. For statement-level triggers, create "`WHEN EXISTS (SELECT * FROM NEW_TABLE` (or `OLD_TABLE) WHERE` (<condition-statement>))"

4 *Build action clause*: Input all the `A.` prefixed entries.

(a) Convert all built-in functions for transition values and aggregate operators into SQL3 format like step 3.(a).

(b) Partition the entries into distinct groups. That is, gather entries with identical variables being used in the same group. Each group will have one data modification statement such as INSERT, DELETE, or UPDATE. Preserve the order among partitioned groups.

(c) For each group G_i, call qbe2sql($< G_i >$) algorithm according to the order in step 4.(b). Let the resulting SQL string for G_i as <action-statement>$_i$. The contents in the statement box are literally copied to <action-statement>$_i$. Then, final action statements for triggers would be "BEGIN ATOMIC <action-statement>$_1$; ..., <action-statement>$_n$; END".

5. TBE AS A UNIVERSAL TRIGGER RULE FORMATION TOOL

At present, TBE supports only SQL3 triggers syntax. Although SQL3 is close to its final form, many database vendors are already shipping their products with their own proprietary triggers syntax. When multiple databases are interconnected or integrating one database to another, these diversities can introduce significant problems. To remedy this problem, one can use TBE as a universal triggers construction tool. The user can create trigger rules using TBE interface and saves them as TBE's internal format. When there is a need to change one database to another, the user can reset the target system (e.g., from Oracle to DB2) to re-generate new trigger rules.

Ideally, we like to be able to add new types of database triggers in a *declarative* fashion. That is, given a new triggers system, a user needs only to describe what kind of syntax the triggers use. Then, TBE should be able to generate the target trigger rules without further intervention from the user. Two inputs to TBE are needed to add new database triggers: *trigger syntax rule* and *trigger composition rule*. In a trigger syntax rule, a detailed description of the syntactic aspect of the triggers is encoded by the declarative language. In a trigger composition rule, information as to how to compose the trigger rule (i.e., English sentence) using the trigger syntax rule is specified. the behavior and output of TBE conforms to the specifics defined in the meta rules of the selected target trigger system. When a user chooses the target trigger system in the interface, corresponding trigger syntax and composition rules are loaded from the meta rule database into TBE system. The high-level overview is shown in Figure 9.

5.1. TRIGGER SYNTAX RULE

TBE provides a declarative language to describe trigger syntax, whose EBNF is shown below:

```
<Trigger-Syntax-Rule> ::= <event-rule> | <condition-rule> | <action-rule>
<event-rule> ::= 'event' 'has' <event-rule-entry> (',' <event-rule-entry>)* ';'
<event-rule-entry> ::= <structure-operation> 'on' ('row' | 'attribute') |
            <activation-time> | <granularity> | <evaluation-time>
<structure-operation> ::= ('I.' | 'D.' | 'U.' | 'RT.') 'as' <value>
<activation-time> ::= ('BFR.' | 'AFT.' | 'ISTD.') 'as' <value>
<granularity> ::= ('R.' | 'S.') 'as' <value>
```

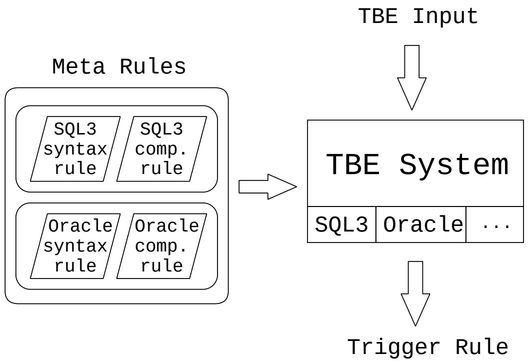

Figure 9 The architecture of TBE as a universal triggers construction tool.

<value> ::= <identifier> | ' <identifier> ' | 'null' | 'true'
<condition-rule> ::= 'condition' 'has' <condition-rule-entry> (',' <condition-rule-entry>)* ';'
<condition-rule-entry> ::= <condition-role> | <condition-context>
<condition-role> ::= 'role' 'as' ('mandatory' | 'optional')
<condition-context> ::= 'context' 'as'
'(' ('NEW | 'OLD | 'NEW_TABLE | 'OLD_TABLE) 'as' <value> ')'
<action-rule> ::= 'action' 'has' <action-rule-entry> (',' <action-rule-entry>)* ';'
<action-rule-entry> ::= <structure-operation> | <evaluation-time>
<evaluation-time> ::= ('DFR.' | 'IMM.' | 'DTC.') 'as' <value>

Although the detailed discussion of the language constructs is beyond the scope of this paper, the essence of the language has the form "command as value", meaning the trigger feature *command* is supported and represented by the keyword *value*. For instance, a clause `NEW_TABLE as INSERTED` for Starburst system would mean that "Starburst supports statement-level triggering and uses the keyword `INSERTED` to access transition values".

Example 8: SQL3 trigger syntax can be described as follows:

```
event has (
    I. as INSERT on row, D. as DELETE on row, U. as UPDATE on attribute,
    BFR. as BEFORE, AFT. as AFTER, R. as ROW, S. as STATEMENT
) ;
condition has (
    role as optional,
    transition as (NEW as NEW, OLD as OLD,
        NEW_TABLE as NEW_TABLE, OLD_TABLE as OLD_TABLE)
) ;
action has (
    I. as INSERT, D. as DELETE, U. as UPDATE
) ;
```

The interpretation of this meta rule should be self-describing. For instance, the fact the there is no clause `S. as ...` implies that SQL3 triggers do not support event monitoring on the selection operation. In addition, the clause `T. as STATEMENT` implies that SQL3 triggers support table-level event monitoring using the keyword 'FOR EACH STATEMENT'.

The partial comparison of the trigger syntax of SQL3, Starburst, Postgres, Oracle and DB2 system is shown in Table 1. Using the language constructs defined above, these syntax can be easily encoded into the trigger syntax rule. Note that our language is limited to the triggers based on ECA and relational data model.

TBE	**SQL3**	**Starburst**	**Postgres**	**Oracle**	**DB2**
I.	INSERT	INSERTED	INSERT	INSERT	INSERT
D.	DELETE	DELETED	DELETE	DELETE	DELETE
U.	UPDATE	UPDATED	UPDATE	UPDATE	UPDATE
RT.	N/A	N/A	RETRIEVE	N/A	N/A
BFR.	BEFORE	N/A	N/A	BEFORE	BEFORE
AFT.	AFTER	true	true	AFTER	AFTER
ISTD.	N/A	N/A	INSTEAD	N/A	N/A
R.	ROW	N/A	TUPLE	ROW	ROW
S.	STATEMENT	true	N/A	true	STATEMENT
NEW	NEW	N/A	NEW	NEW	NEW
OLD	OLD	N/A	CURRENT	OLD	OLD
NEW_TABLE	NEW_TABLE	INSERTED, NEW-UPDATED	N/A	N/A	NEW_TABLE
OLD_TABLE	OLD_TABLE	DELETED, OLD-UPDATED	N/A	N/A	OLD_TABLE

Table 1 Syntax comparison of five triggers using the *trigger syntax rule*. The leftmost column contains TBE commands while other columns contain equivalent keywords of the corresponding trigger system. "N/A" means the feature is not supported and "true" means the feature is supported by default.

5.2. TRIGGER COMPOSITION RULE

After the syntax is encoded, TBE still needs information on how to compose English sentences for trigger rules. This logic is specified in the trigger composition rule. In a trigger composition rule, a macro variable is surrounded by the `$` sign and substituted with actual values during rule generation time.

Example 9: The following is a SQL3 trigger composition rule:

```
CREATE TRIGGER $trigger-name$
    $activation-time$ $structure-operation$ ON $table$
    FOR EACH $granularity$
    WHEN $condition-statement$
    BEGIN ATOMIC
        $action-statement$
```

```
END
```

In rule generation time, for instance, variable \$activation-time\$ is replaced with value either `BEFORE` or `AFTER` since those two are only valid values according to the trigger syntax rule in Example 8. In addition, variables \$condition-statement\$ and \$action-statement\$ are replaced with statements generated by the translation algorithm in Section 4.2.

6. RELATED WORK

Past active database research has focused on active database rule languages (Agrawal and Gehani, 1989), rule execution semantics (Cochrane et al., 1996), or rule management and system architecture issues (Simon and Kotz-Dittrich, 1995). In addition, research on visual querying has been done in traditional database research (Embley, 1989),(Zloof, 1977). To a greater or lesser extent, all these research focused on devising novel visual querying schemes to replace data retrieval aspects of SQL language. Although some have considered data definition aspects (Collet and Brunel, 1992) or manipulation aspects, none have extensively considered the *trigger* aspects of SQL, especially from the user interface point of view.

Other work (e.g., IFO_2 (Teisseire et al., 1994), IDEA (Ceri et al., 1996) have attempted to build graphical triggers description tools, too. Using IFO_2, one can describe how different objects interact through events, thus giving priority to an overview of the system. Argonaut from the IDEA project (Ceri et al., 1996) focused on the automatic generation of active rules that correct integrity violation based on declarative integrity constraint specification and active rules that incrementally maintain materialized views based on view definition. TBE, on the other hand, tries to help users *directly* design active rules with minimal learning.

Other than QBE skeleton tables, *forms* have been popular building blocks for visual querying mechanism as well. For instance, Embley, 1989) proposes the NFQL as a communication language between humans and database systems. It uses forms in a strictly nonprocedural manner to represent query. Other work using forms focused on the querying aspect of the visual interface (Collet and Brunel, 1992). To the best of our knowledge, the only work that is directly comparable to ours is RBE (Chang and Chen, 1997). TBE is different from RBE in the following aspects:

- Since TBE is designed with SQL3 triggers in mind, it is capable of creating all the complex SQL3 trigger rules. Since RBE's capability is limited to OPS5-style production rules, it cannot express the subtle difference of the trigger activation time nor granularity.

- The implementation of RBE is tightly coupled with the underlying rule system and database so that it cannot easily support multiple heterogeneous database triggers. Since TBE implementation is a thin layer utilizing a translation from a visual representation to the underlying triggers, it is loosely coupled with the database.

7. CONCLUSION

In this paper, we presented the design and implementation of TBE, a visual trigger rule specification interface. QBE was extended to handle features specific to ECA trigger rules. TBEextends the visual querying mechanism from QBE and applies it to triggers construction application. Examples to demonstrate SQL3-based trigger rule generation procedure as well as the TBE to SQL3 trigger translation algorithm were given. Extensions to make TBE a universal trigger rule interface was also discussed. For a trigger system s, we can declaratively specify the syntax mapping between TBE and s, so that we can use TBE not only as a trigger rule formation tool, but also a universal intermediary for translations between supported systems.

References

Agrawal, R. and Gehani, N. (1989) Ode (Object Database and Environment): The Language and the Data Model, *Proc. SIGMOD*.

Codd, E. F. (1972) Relational Completeness of Data Base Languages, *Data Base Systems, Courant Computer Symposia Series*, Prentice-Hall, 6:65-98.

Collet, C. and Brunel, E. (1992) Definition and Manipulation of Forms with FO2, *Proc. IFIP Visual Database Systems*.

Chang, Y.-I. and Chen, F.-L. (1997) RBE: A Rule-by-example Action Database System, *Software – Practice and Experience*, 27(4):365-394.

Ceri, S., et al (1996) Active Rule Management in Chimera, In J. Widom and S. Ceri (ed.) *Active Database Systems: Triggers and Rules for Active Database Processing*, Morgan Kaufmann.

Cochrane, R., Pirahesh, H. and Mattos, N. (1996) Integrating Triggers and Declarative Constraints in SQL Database Systems, *Proc. VLDB*.

Embley, D. W. (1989) NFQL: The Natural Forms Query Language, *ACM TODS*, 14(2):168-211.

Embury, S. M. and Gray, P. M. D. (1998) Database Internal Applications, In N. W. Paton (ed.), *Active Rules In Database Systems*, Springer-Verlag.

Gatziu, S. and Dittrich, K. R. (1998) SAMOS, In N. W. Paton (ed.), *Active Rules In Database Systems*, Springer-Verlag.

Lee, D., Mao, W. and Chu, W. W. (1999) TBE: Trigger-By-Example, UCLA-CS-TR-990029, (http://www.cs.ucla.edu/~dongwon/paper/).

McLeod, D. (1976) The Translation and Compatibility of SEQUEL and Query by Example, *Proc. Int'l Conf. Software Engineering*, CA.

Melton, J. (ed.) (1999) (ANSI/ISO Working Draft) SQL/Foundation, *ANSI X3H2-99-079/WG3:YGJ-011*, March.

Paton, N. W. (ed.) (1998) Active Rules in Database Systems, *Springer-Verlag*.

Simon, E. and Kotz-Dittrich, A. (1995) Promises and Realities of Active Database Systems, *Proc. VLDB*.

Teisseire, M., et al. (1994) Towards Event-Driven Modelling for Database Design, *Proc. VLDB*.

Zloof, M. M. (1977) Query-by-Example: a data base language, *IBM System J.*, 16(4):342-343.

Zaniolo, C., et al. (1997) Advanced Database Systems, *Morgan Kaufmann*.

Biographies

Dongwon Lee received his B.S. from Korea Univ. in 1993 and M.S. from Columbia Univ. in 1995. He is currently a Ph.D student at UCLA. Prior to that, he has worked at AT&T Bell Labs., Murray Hill, NJ (now AT&T Labs – Research) for two years.

Wenlei Mao is currently a Ph.D student in Computer Science Department of UCLA. He received his M.S. in Computer Science from College of William and Mary in 1996, and B.S. in Physics from Peking Univ. in 1992. His research interests include active and temporal databases.

Henry Chiu is a recent graduate from UCLA with a B.S. in Computer Science and Engineering, now entering the M.S. program. His interests include developing graphical interfaces and tools.

Wesley W. Chu is a professor and the past chairman (1988-1991) of the Computer Science Department at the University of California, Los Angeles. His current research interests are in the area of intelligent information systems and medical information systems. Dr. Chu is a Fellow of IEEE and the associate editor for the J. of VLDB and J. of Data and Knowledge Engineering.