TBE: Trigger-By-Example*

Dongwon Lee, Wenlei Mao, and Wesley W. Chu

Department of Computer Science
University of California, Los Angeles
Los Angeles, CA 90095, USA
{dongwon,wenlei,wwc}@cs.ucla.edu

Abstract. TBE (Trigger-By-Example) is proposed to assist users in writing trigger rules. TBE is a graphical trigger rule specification language and system to help users understand and specify active database triggers. Since TBE borrowed its basic idea from QBE, it retained many benefits of QBE while extending the features to support triggers. Hence, TBE is a useful tool for novice users to create simple trigger rules easily. Further, since TBE is designed to insulate the details of underlying trigger systems from users, it can be used as a universal trigger interface for rule formation.

1 Introduction

Triggers provide a facility to autonomously react to events occurring on the data, by evaluating a data-dependent condition, and by executing a reaction whenever the condition evaluation yields a truth value. Such triggers have been adopted 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 its wide deployment is the lack of tools that aid users to create complex trigger rules in a simple manner. In many environments, the correctness of written trigger rules is very crucial since the semantics encoded in the trigger rules are shared by many applications [18]. 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.

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 domain relational calculus, its expressive power has proved to be equivalent to that of SQL which is based on tuple relational calculus [2]. As opposed to SQL where users have to conform to the phrase structure strictly, QBE users 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 [17].

In this paper, we propose to use the established QBE as a user interface for writing trigger rules. Since most trigger rules are complex combinations of SQL

* This research is supported in part by DARPA contract No. N66001-97-C-8601 and SBIR F30602-99-C-0106.

statements, by using QBE as a user interface for triggers, the user may create only admissible trigger rules. The main idea is to use QBE in a *declarative* fashion for writing the *procedural* trigger rules [6].

2 Background and Related Work

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 [13].

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 unquoted. Let us see a QBE example. We use the following schema throughout the paper.

Example 1. 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 2 shows two equivalent representations of the query in SQL3 and QBE, respectively.

Example 2. 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

Related Work Past active database research has focused on active database rule language (e.g., [1]), rule execution semantics (e.g., [6]), or rule management and system architecture issues (e.g., [15]). In addition, research on visual querying has been done in traditional database research (e.g., [7, 17]). 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 [3] or manipulation aspects, none have extensively considered the *trigger* aspects for SQL, especially from the user interface point of view.

Other works (e.g., IFO_2 [16], IDEA [5]) have attempted to build graphical triggers description tools. 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 [5] focused on the automatic generation of active rules that correct integrity violations based on declarative integrity constraint specifications, and active rules that incrementally maintain materialized views based on view definitions. TBE, on the other hand, helps users to *directly* design active rules with minimal learning.

Other than QBE skeleton tables, *forms* have been popular building blocks for visual querying mechanisms as well. For instance, [7] proposes the NFQL as a communication language between humans and database systems. It uses forms in a strictly nonprocedural manner to represent queries. Other works using forms are mostly for querying aspect of the visual interface [3].

To the best of our knowledge, the only work that is directly comparable to ours is RBE [4]. Although RBE also uses the idea of QBE as an interface for creating trigger rules, there are the following differences:

- Since TBE is carefully designed with SQL3 triggers in mind, it is capable of creating all the complex SQL3 trigger rules. Since RBE's capability is, however, limited to OPS5-style production rules, it cannot express, for instance, the subtle difference of the trigger activation time nor granularity.
- No evident suggestion of RBE as a user interface for writing triggers is given. On the other hand, TBE is specifically aimed for that purpose.
- 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.

The organization of this paper is as follows. Section 3 proposes our TBE for SQL3 triggers and discusses several related issues. A few complex SQL3 trigger examples are illustrated in Section 4. The preliminary implementation and potential applications of TBE are presented in Sections 5 and 6, respectively. Concluding remarks are given in Section 7.

3 TBE: Trigger-By-Example

We propose to use QBE as a user interface for writing trigger rules. Our tool is called Trigger-By-Example (TBE) which has the same spirit as that of QBE. The philosophy of QBE is to require the user to know very little in order to get started and to minimize the number of concepts that he or she subsequently has to learn to understand and use the whole language [17]. By using QBE as an interface, we attain the same benefits for creating trigger rules.

3.1 Difficulty of Expressing Procedural Triggers in Declarative QBE

Triggers in SQL3 are procedural in nature. Trigger actions can be arbitrary SQL procedural statements, allowing not only SQL data statements (i.e., select, project, join) but also transaction, connection, session statements[1]. Also, the order among action statements needs to be obeyed faithfully to preserve the correct semantics. On the contrary, QBE is a declarative query language. While writing a query, the user does not have to know if the first row in skeleton tables needs to be executed before the second row or not. That is, the order is immaterial. Also 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. Therefore, QBE cannot really handle other procedural SQL statements such as transaction or user-defined functions in a simple manner. 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.

In what follows, we shall describe how QBE was extended to be TBE, what design options were available, and which option was chosen by what rationale, etc.

3.2 Trigger Name

A unique name for each trigger rule needs to be set in a special input box, called the *name box*, where the user can fill in an arbitrary identifier as shown below:

<TriggerRuleName>

Typically, the user first decides the trigger name and then proceeds to the subsequent tasks. There are often cases when multiple trigger rules are written together in a single TBE query. For such cases, the user needs to provide a unique trigger name for each rule in TBE query separately. In what follows, when there is only a single trigger rule in the example, we take the liberty of not showing the trigger name for briefness.

3.3 Event-Condition-Action Triggers

SQL3 triggers use the ECA model. Therefore, triggers are represented by mainly three isolated E, C, A parts. In TBE, each E, C, A part maps to the corresponding skeleton tables separately. To differentiate among three parts, three prefix flags, `E.`, `C.`, `A.`, are introduced. That is, in skeleton tables, table name is prefixed with one of these flags. The condition box in QBE is also similarly extended. For instance, a condition statement is specified in the `C.` prefixed skeleton table and condition box below.

C.emp	Eno	Ename	DeptNo	Sal

C.conditions

[1] SQL3 triggers definition in [13] leaves it implementation-defined whether the transaction, connection, or session statements should be contained in the action part or not.

3.4 Triggers Event Types

SQL3 triggers allow only the INSERT, DELETE, and UPDATE as legal event types. Coincidentally, QBE has constructs `I.`, `D.`, and `U.` for each event type to describe the data manipulation query. The TBE uses these constructs to describe the trigger event types. Since the INSERT and DELETE always affect the whole tuple rather than individual columns, `I.` and `D.` must be filled in the leftmost column of skeleton table. When the UPDATE trigger is described as to particular column, then `U.` is filled in the corresponding column. Otherwise, `U.` is filled in the leftmost column. Consider the following example.

Example 3. 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 that since SQL3 triggers definition limits that only a *single* event be monitored per *single* rule, there can not be more than one row having `I.`, `D.`, or `U.` flag unless multiple trigger rules are written together. Therefore, same trigger actions for different events (e.g., "abort when either INSERT or DELETE occurs") need to be expressed as separate trigger rules in SQL3 triggers.

3.5 Triggers Activation Time and Granularity

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

1. The activation time can have two modes, *before* and *after*. The *before* mode triggers execute before their event and are useful for conditioning the input data. The *after* mode triggers execute after their event and are typically used to embed application logic [6]. In TBE, two corresponding constructs, `BFR.` and `AFT.`, are introduced to denote these modes. The "." is appended to denote that these are built-in system commands.
2. The granularity of a trigger can be specified as either *for each row* or *for each statement*, referred to as *row-level* and *statement-level* triggers, respectively. The row-level triggers are executed after each modification to tuple whereas the statement-level triggers are executed once for an event regardless of the number of the tuples affected. In TBE notation, `R.` and `S.` are used to denote the row-level and statement-level triggers, respectively.

Consider the following illustrating example.

Example 4. SQL3 and TBE representation for a trigger with *after* activation time and *row-level* granularity.

CREATE TRIGGER **AfterRowLevelRule**
AFTER UPDATE OF **Ename, Sal** ON **emp** FOR EACH ROW

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

3.6 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`) or tables (i.e., `OLD_TABLE`, `NEW_TABLE`) depending on the type of triggers, whether row-level or statement-level. 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 and tables. We have considered the following two approaches to introduce the transition values in TBE.

1. *Using new built-in functions*: 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()` return a single tuple with value before and after the change, respectively. Therefore, applying aggregate functions such as `CNT.` or `SUM.` to the `OLD()` or `NEW()` is meaningless (i.e., `CNT.NEW(_s)` is always 1 or `SUM.OLD(_s)` is always same as `_s`). Using new built-in functions, for instance, the 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

   Also the event "when salary is doubled for each row" can be represented as follows:

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

E.conditions
NEW(_s) > OLD(_s) * 2

   It is illegal to apply the `NEW()` or `NEW_TABLE()` to the variable defined on the DELETE event. Likewise for the application of `OLD()` or `OLD_TABLE()` to the variable defined on the INSERT event. Asymmetrically, it is redundant to apply the `NEW()` or `NEW_TABLE()` the variable defined on the INSERT event. Likewise for the application of `OLD()` or `OLD_TABLE()` to the variable defined on the DELETE event. For instance, in the above event "every time more than 10 new employees are inserted", `_n` and `NEW_TABLE(_n)` are equivalent. Therefore, the condition expression at the condition box can be rewritten as "`CNT.ALL._n` > 10". It is ambiguous, however, to simply refer to the variable defined in the UPDATE event without the built-in functions. That is, in the event "when salary is doubled for each row", `_s` can refer to values both before and after the UPDATE. That is, "`_s > _s * 2`" at the condition box would cause an error due to its ambiguity. Therefore, for the UPDATE event case, one needs to explicitly use the built-in functions to access transition values.

2. *Using modified skeleton tables*: Depending on the event type, skeleton tables are modified accordingly; additional columns may appear in the skeleton tables[2]. For the INSERT event, a keyword `NEW_` is prepended to the existing column names in the skeleton table to denote that these are newly inserted ones. For the DELETE event, a keyword `OLD_` is prepended similarly. For the UPDATE event, a keyword `OLD_` is prepended to the existing column names whose values are updated in the skeleton table to denote values before the UPDATE. At the same time, additional columns with a keyword `NEW_` appear to denote values after the UPDATE. If the UPDATE event is for all columns, then `OLD_`column-name and `NEW_`column-name appear for all columns.
Consider an event "when John's salary is doubled within the same department". Here, we need to monitor two attributes – Sal and DeptNo. First, the user may type the event activation time and granularity information at the leftmost column as shown in the first table. Then, the skeleton table changes its format to accommodate the UPDATE event effect as shown in the second table. That is, two more columns appear and the `U.` construct is relocated to the leftmost column.

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

E.emp	Eno	Ename	OLD_DeptNo	NEW_DeptNo	OLD_Sal	NEW_Sal
AFT.U.R.						

Then, the user fills in variables into the proper columns to represent the conditions. For instance, "same department" is expressed by using same variable `_d` in both OLD_DeptNo and NEW_DeptNo columns.

[2] We have also considered modifying tables, instead of columns. For instance, for the INSERT event, a keyword `NEW_` is prepended to the *table* name. For the UPDATE event, a keyword `OLD_` is prepended to the *table* name while new table with a `NEW_` prefix is created. This approach, however, was not taken because we wanted to express column-level UPDATE event more explicitly. That is, for an event "update occurs at column Sal", we can add only OLD_Sal and NEW_Sal attributes to the existing table if we use the "modifying columns" approach. If we take the "modifying tables" approach, however, we end up with two tables with all redundant attributes whether they are updated or not (e.g., two attributes OLD_emp.Ename and NEW_emp.Ename are unnecessarily created; one attribute emp.Ename is sufficient since no update occurs for this attribute).

E.emp	Eno	Ename	OLD_DeptNo	NEW_DeptNo	OLD_Sal	NEW_Sal
AFT.U.R.		John	_d	_d	_o	_n

E.conditions
_n > _o * 2

We chose the approach using new built-in functions to introduce transition values into TBE. Although there is no difference with respect to the expressive power between two approaches, the first one does not incur any modifications to the skeleton tables, thus minimizing cluttering of the user interface.

3.7 The `REFERENCING` Construct

SQL3 allows the renaming of transition variables or tables using the `REFERENCING` construct for the user's convenience. In TBE, this construct is not needed since the transition values are directly referred to by the variables filled in the skeleton tables.

3.8 Procedural Statements

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, and thus TBE described so far, can achieve, we instead provide a special kind of box, called *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. For example,

A.statements
IF (X > 10) ROLLBACK;

3.9 The Order among Action Trigger Statements

SQL3 allows multiple action statements in triggers, each of which is executed according to the order they are written. To represent triggers whose semantics depend on the assumed sequential execution, TBE uses an implicit agreement; like prolog, the execution order follows from top to bottom. Special care needs to be taken in translation time for such action statements as follows:

- The *action* skeleton tables appearing before are translated prior to that appearing after.
- In the same *action* skeleton tables, action statements written at the top row are translated prior to that written at the bottom one.

3.10 Expressing Conditions in TBE

In most active database triggers languages, the event part of the triggers language is exclusively concerned with what has happened and cannot perform tests on values associated with the event. Some triggers languages (e.g., Ode [1], SAMOS [9], Chimera [5]), however, provide filtering mechanisms that perform tests on event parameters (see [14], chapter 4). Event filtering mechanisms can be very useful in optimizing trigger rules; only events that passed the parameter filtering tests are sent to the condition module to avoid unnecessary expensive condition evaluations.

In general, we categorize condition definitions of the triggers into 1) *parameter filter (PF)* type and 2) *general constraint (GC)* type. SQL3 triggers definition does not have PF type; event language specifies only the event type, activation time and granularity information, and all conditions (both PF and GC types) need to be expressed in the `WHEN` clause. In TBE, however, we decided to allow users to be able to differentiate PF and GC types by providing separate condition boxes (i.e., `E.` and `C.` prefixed ones) although it is not required for SQL3. This is because we wanted to support other trigger languages who have both PF and GC types in future.

1. *Parameter Filter Type*: Since this type tests the event parameters, the condition must use the transition variables or tables. Event examples such as "every time more than 10 new employees are inserted" or "when salary is doubled" in Section 3.6 are these types. In TBE, this type is typically represented in the `E.` prefixed condition box.
2. *General Constraint Type*: This type expresses general conditions regardless of the event type. In TBE, this type is typically represented in the `C.` prefixed condition boxes. One such example is illustrated in Example 5.

Example 5. When an employee's salary is increased more than twice within the same year (a variable CURRENT_YEAR contains the current year value), record changes into the `log(Eno, Sal)` table. Assume that there is another table `sal-change(Eno, Cnt, Year)` to keep track of the employee's salary changes.

```
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
```

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

C.sal-change	Eno	Cnt	Year
	NEW(_n)	_c	CURRENT_YEAR

C.conditions
_c >= 2

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

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

Here, the condition part of the trigger rule (i.e., `WHEN` clause) checks the Cnt value of the sal-change table to check how many times salary was increased in the same year, and thus, does not involve testing any transition values. Therefore, it makes more sense to represent such condition as GC type, not PF type. Note that the headers of the sal-change and condition box have the `C.` prefixes.

4 Complex SQL3 Triggers Examples

In this section, we show a few complex SQL3 triggers and their TBE representations. These trigger examples are modified from the ones in [8, 18].

4.1 Integrity Constraint Triggers

A trigger rule to maintain the foreign key constraint is shown below.

Example 6. 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 E1 WHERE E1.DeptNo =
      (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 that how _e variable is used to join the emp and dept tables to find the department whose manager is just deleted. 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	

Another example is shown below.

Example 7. 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 same TBE query would represent different SQL3 triggers. That is, row-level triggers 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
```

We believe that this is a good example illustrating why TBE is useful in writing trigger rules. That is, when the only difference between two rules is the trigger granularity, simple change between `R.` and `S.` is sufficient in TBE. However, in SQL3, users should devise quite different rule syntaxes as demonstrated above.

4.2 View Maintenance Triggers

Suppose a company maintains the following view derived from the emp and dept schema.

Example 8. Create a view `HighPaidDept` that has at least one "rich" employee earning more than 100K.

```
CREATE VIEW HighPaidDept AS
      SELECT DISTINCT D.Dname
      FROM emp E, dept D
      WHERE E.DeptNo = D.Dno AND E.Sal > 100K
```

The straightforward way to maintain the views upon changes to the base tables is to re-compute all views from scratch. Although incrementally maintaining the view is more efficient than this method, for the sake of trigger example, let us implement the naive scheme below. The following is only for UPDATE event case.

Example 9. Refresh the `HighPaidDept` when UPDATE occurs on emp table.

```
CREATE TRIGGER RefreshView AFTER UPDATE OF DeptNo, Sal ON emp
FOR EACH STATEMENT
```

```
BEGIN ATOMIC
   DELETE FROM HighPaidDept;
   INSERT INTO HighPaidDept
      (SELECT DISTINCT D.Dname FROM emp E, dept D
          WHERE E.DeptNo = D.Dno AND E.Sal > 100K);
END
```

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

A.emp	Eno	Ename	DeptNo	Sal
			_d	> 100K

A.dept	Dno	Dname	MgrNo
	_d	_n	

A.HighPaidDept	Dname
D.	
I.	_n

By the implicit ordering of TBE, the DELETE statement executes prior to the INSERT statement.

5 Implementation

A preliminary version of the TBE prototype has been implemented using jdk 1.2. Although the underlying concept is the same as what we have presented so far, we added several bells and whistles (e.g., context sensitive pop-up menu) for better human-computer interaction. The algorithm to generate trigger rules from TBE is omitted due to space limitation. For details, please refer to [10].

The main screen consists of two sections – one for input and another for output. The input section is where the user creates trigger rules by QBE mechanism and the output section is where the interface generates trigger rules in the target trigger syntax. Further, the input section consists of three panes for event, condition, action, respectively. The main screen of the prototype is shown in Figure 1, where the query in Example 5 is shown.

6 Applications

Not only is TBE useful for writing trigger rules, but it can also be used for other applications with a few modifications. Two such applications are illustrated in this section.

Declarative Constraints in SQL3: SQL3 has the `ASSERTION` to enforce any condition expression that can follow `WHERE` clause to embed some application logic. The syntax of the `ASSERTION` is:

CREATE ASSERTION **<assertion-name>** CHECK **<condition-statement>**

Note the similarity between the assertion and triggers syntax in SQL3. Therefore, a straightforward extension of TBE can be used as a tool to enforce assertion constraints declaratively. In fact, since the `ASSERTION` in SQL3 only permits declarative constraints, TBE suits the purpose perfectly.

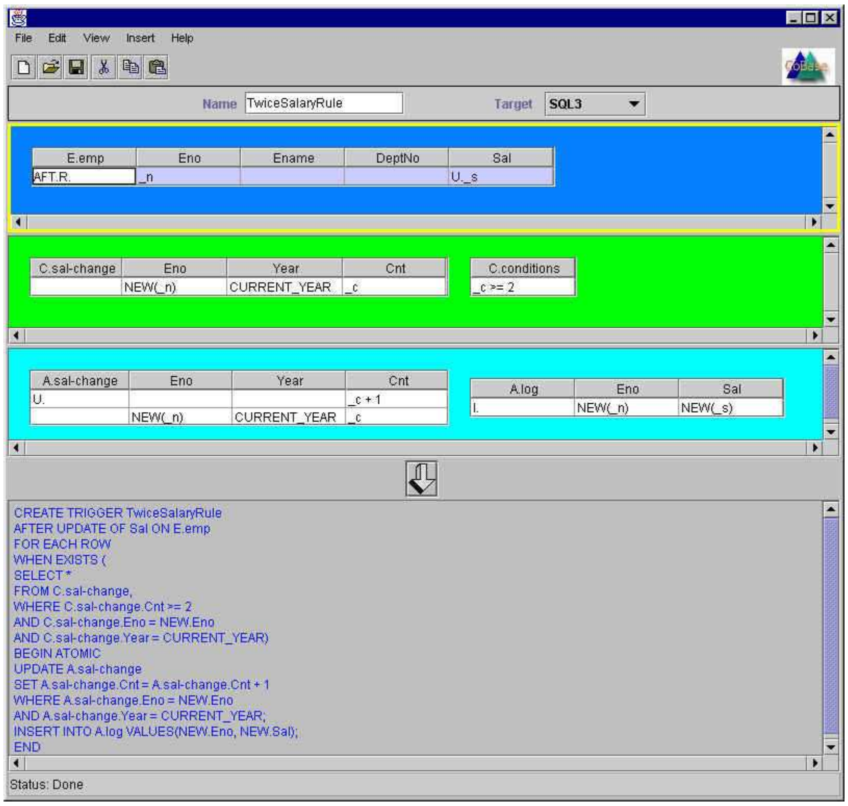

Fig. 1. Main screen dump.

Universal Triggers Construction Tool: Although SQL3 is close to its final form, many database vendors are already shipping their products with their own proprietary trigger syntaxes and implementations. When multiple databases are used together or one database needs to be migrated to another, these diversities can introduce significant problems. To remedy this problem, one can use TBE as a universal triggers construction tool. That is, the user creates triggers using TBE interface.When changing database from one to another (e.g., from Oracle to DB2), the user can simply reset one of the preference information of TBE to re-generate the new trigger rules. Extending TBE to support all unique features of diverse database products is not a trivial task. Nevertheless, we believe that retaining the visual nature of the triggers construction with TBE can be useful in coping with heterogeneous database systems.

7 Conclusion

A novel user interface called TBE for creating triggers is proposed. TBE borrows the visual querying mechanism from QBE and applies it to triggers construction application in a seamless fashion. An array of new constructs are introduced to extend QBE to support triggers semantics and syntaxes properly. To prove the concept, a prototype is implemented and demonstrated the feasibility and benefits of applying QBE in writing trigger rules.

References

1. Agrawal, R., Gehani, N. "Ode (Object Database and Environment): The Language and the Data Model", *Proc. SIGMOD*, Portland, Oregon, 1989.
2. Codd, E. F. "Relational Completeness of Data Base Languages", *Data Base Systems, Courant Computer Symposia Series*, Prentice-Hall, 6:65-98, 1972.
3. Collet, C., Brunel, E. "Definition and Manipulation of Forms with FO2", *Proc. IFIP Visual Database Systems*, 1992.
4. Chang, Y.-I., Chen, F.-L. "RBE: A Rule-by-example Action Database System", *Software – Practice and Experience*, 27(4):365-394, 1997.
5. Ceri, S., Fraternali, P., Paraboschi, S., Tanca, L. "Active Rule Management in Chimera", In J. Widom and S. Ceri (ed.), *Active Database Systems: Triggers and Rules for Active Database Processing*, Morgan Kaufmann, 1996.
6. Cochrane, R., Pirahesh, H., Mattos, N. "Integrating Triggers and Declarative Constraints in SQL Database Systems", *Proc. VLDB*, 1996.
7. Embley, D. W. "NFQL: The Natural Forms Query Language", *ACM TODS*, 14(2):168-211, 1989.
8. Embury, S. M., Gray, P. M. D. "Database Internal Applications", In N. W. Paton (ed.), *Active Rules In Database Systems*, Springer-Verlag, 1998.
9. Gatziu, S., Dittrich, K. R. "SAMOS", In N. W. Paton (ed.), *Active Rules In Database Systems*, Springer-Verlag, 1998.
10. Lee, D., Mao, W., Chiu, H., Chu, W. W. "TBE: A Graphical Interface for Writing Trigger Rules in Active Databases", *5th IFIP 2.6 Working Conf. on Visual Database Systems (VDB)*, 2000.
11. Lee, D., Mao, W., Chu, W. W. "TBE: Trigger-By-Example (Extended Version)", *UCLA-CS-TR-990029*, 1999. http://www.cs.ucla.edu/~dongwon/paper/
12. McLeod, D. "The Translation and Compatibility of SEQUEL and Query by Example", *Proc. Int'l Conf. Software Engineering*, San Francisco, CA, 1976.
13. Melton, J. (ed.), "(ANSI/ISO Working Draft) Foundation (SQL/Foundation)", *ANSI X3H2-99-079/WG3:YGJ-011*, March, 1999. ftp://jerry.ece.umassd.edu/isowg3/dbl/BASEdocs/public/sql-foundation-wd-1999-03.pdf
14. Paton, N. W. (ed.), "Active Rules in Database Systems", *Springer-Verlag*, 1998.
15. Simon, E., Kotz-Dittrich, A. "Promises and Realities of Active Database Systems", *Proc. VLDB* 1995.
16. Teisseire, M., Poncelet, P., Cichetti, R. "Towards Event-Driven Modelling for Database Design", *Proc. VLDB*, 1994.
17. Zloof, M. M. "Query-by-Example: a data base language", *IBM System J.*, 16(4):342-343, 1977.
18. Zaniolo, C., Ceri, S., Faloutsos, C., Snodgrass, R. R., Subrahmanian, V.S., Zicari, R. "Advanced Database Systems", *Morgan Kaufmann*, 1997.