VTT INFORMATION TECHNOLOGY Research Report No. J-4

Antoni Wolski

Timer and Counter Triggers in RAPID

December 1995

Espoo

VTT INFORMATION TECHNOLOGY P.O. Box 1201 FIN-02044 VTT, Finland Tel. (-358 0) 4561 Fax (+358 0) 456 6027

Timer and Counter Triggers in RAPID

Antoni Wolski

VTT Information Technology
P.O. Box 1201, 02044 VTT, Finland

e-mail: wolski@vtt.fi

Abstract

Triggers are event-condition-action rules in active databases. In order to improve the power of expression of triggers, the event part is typically modeled as a composite event or an event expression. We propose to model events as finite state automata of pre-defined types. We show how two example triggers types may be defined using a trigger definition language based on SQL3. The timer trigger is a trigger based on the timer automaton, and the counter trigger is based on the counter automaton. The approach results in a intuitive and simple triggers definition syntax and it allows for a stepwise trigger system implementation in object-oriented environments. The discussed triggers were implemented in the RAPID system–a temporal and active fast-response database system for industrial applications.

1 Introduction

The general interest in active databases has its source in the perception that a great deal of dynamic characteristics of an application model can be effectively captured as ECA (event-condition-action) [MCD89] rules. Usually ECA rules, are implemented as *triggers* which are easy-to-maintain database objects. An ECA rule applied to relational databases, may be expressed, as in the SQL3 language [SQL3], in the following way:

```
trigger-definition::=
        CREATE TRIGGER trigger-name
        event-type
        ON table-name
        [WHEN (trigger-condition)]
        (action)
```

December 1st, 1995.

The meaning of the above definition is that the procedural part action is executed if, at the point an event of the type event-type occurs in connection with table table-name and the condition trigger-condition is evaluated to true. If the condition is not specified, the action is executed unconditionally at the event occurrence.

In active systems of the first generation [WC959], the event-type corresponded to elementary database modifying operations, like SQL commands UPDATE, DELETE and INSERT. Various application areas, e.g. computer integrated manufacturing and market trading, required triggers of greater expressive power, being able to use complex events defined over histories of events. Such higher-level events are called composite events or event expressions [GJS92]. Several formal languages have been proposed for expressing composite events; see, e.g. [CM93, GJS92, GD94]. The proposed languages are of general nature, and no practical language has been proposed to be embedded into a database language.

In the RAPID project (1992-95), we were developing a database system of the ARCS (Active Rapidly-Changing-Data System) [Dat94] type, i.e. a system designed to accommodate a high insert rate of sensor data and having a capability of active behavior governed by state transitions in the data.

The requirements analysis [JP93] performed during the project among potential industrial users of the system revealed that basic, first-generation triggers would not be sufficient. Capabilities to recognize more complex events were needed, most notably events related to expiration of delay.

It has been proposed [GJS92] to model composite events as finite state automata having a special "accepting" state. Upon entering this state, the composite event "fires", i.e. stimulates the condition evaluation and the subsequent action execution.

One problem with the above approach is that a specification of a general purpose automaton requires a complex notation which is not intuitive for an end-user. Our experience from the RAPID needs survey was that the required composite event capabilities can be classified into certain automaton types. For each of these automaton types a specification notation may be designed, resulting in a simple and intuitive notation. The RAPID trigger definition is based on this principle. Two special-purpose automaton types were implemented: timers and counters.

A minor departure from the model of [GJS92] is that there is no "accepting" state in the automaton. Instead, some transitions to the idle (initial) state results in a "firing" of a trigger. Because, in the same time, the trigger assumes the initial state, no additional transition is necessary after the firing event.

2 General format of the RAPID trigger definition

The syntax shown in the Introduction has been slightly modified:

```
trigger-definition::=
        CREATE TRIGGER trigger-name
        event-type
        ON table-name
        [event-specification]
        [WHEN (trigger-condition)]
        (action-list)

event-type ::=
        {INSERT
        | {UPDATE [OF column-name ...]}
        | WRITE
        | TIMER
        | COUNTER}
```

Where TIMER and COUNTER represent composite event automaton types, and event-specification contains type-specific automaton definition:

```
event-specification::= timer-specification | counter-specification
```

The action part has the following format:

```
action-list::= action-name@action-server-name
        [, action-name@action-server-name]…
```

action-name ::= *a name of an external procedure*

action-server-name::= *network address of a process performing the action*

Actions in RAPID are thus applications procedures [WC959]. The event-condition coupling is immediate and condition-action coupling is decoupled [WC959].

3 Timer triggers

Delay is a pre-defined time span [Jen+94]. A delay may be attached to an instant, yielding an interval. Expressing processing rules in terms of delays is commonplace in control systems. A typical example is:

> **Example 1**
> "If the temperature exceeds the alarm activation level 90, activate the overheating alarm if the temperature does not decline below the clear level 80, during the following 10 minutes."

The above rule, typical for controlling processes with hysteresis, can be illustrated in the following picture:

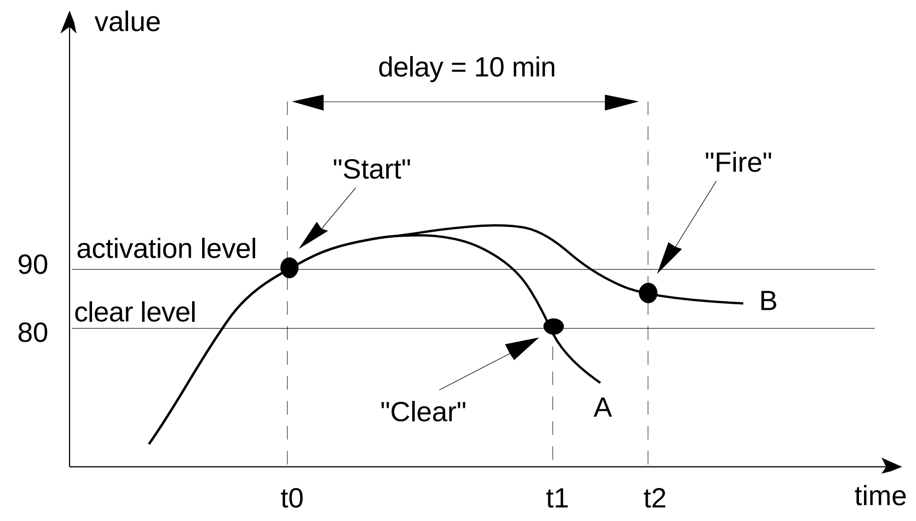

Fig. 1. Applying delay to controlling a process with hysteresis. In case A, the pending alarm is cleared at instant t1. In case B, the alarm is activated at instant t2.

Delays in RAPID are managed by way of a timer automaton specified in the following way:

```
timer-specification::=
        SET interval-literal
        START ON INSERT { (trigger-condition) | ALWAYS }
        [CLEAR ON INSERT { (trigger-condition) | ALWAYS } ]
        [CLEAR NOW]
```

The idea of the above syntax is that a state transition (START or CLEAR) is activated by an elementary event (INSERT) guarded by the trigger-condition.

The state behavior of the timer may be summarized using a state diagram. We assume the following values may be associated with the start and clear conditions:

Start condition	Clear condition	
ST=a	CL=a	- true by definition (ALWAYS)
ST=t	CL=t	- true by evaluation
ST=f	CL=f	- false by evaluation
	CL=Ø	- empty (CLEAR omitted)

All possible state transitions of a timer are shown in Fig. 2. The notation of state transitions includes the controlling event (INSERT or Timeout) and the composite condition under which the transition occurs, e.g. INSERT:(ST=t)•(CL≠t) means "the transition takes place when an INSERT occurs, the start condition is true by evaluation **and** the clear condition has any value except true by evaluation".

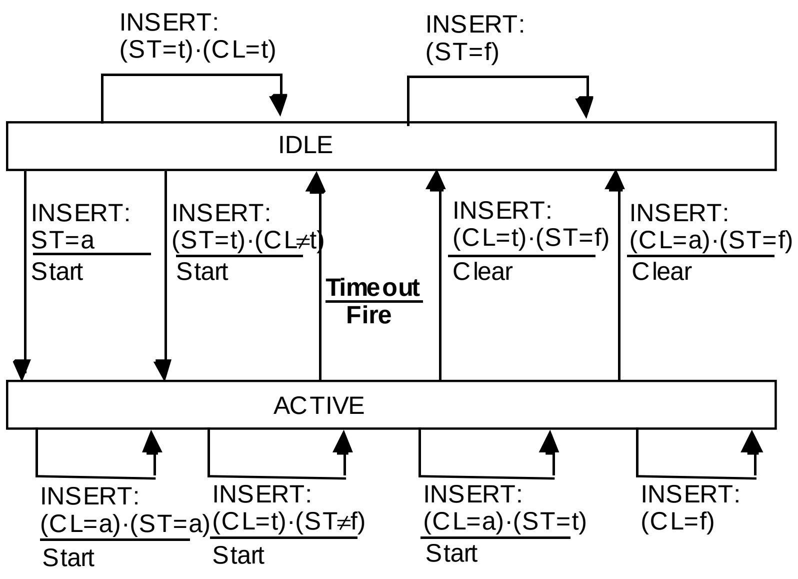
INSERT: (ST=t)·(CL=t)
INSERT: (ST=f)
IDLE
INSERT: ST=a / Start
INSERT: (ST=t)·(CL≠t) / Start
Timeout / Fire
INSERT: (CL=t)·(ST=f) / Clear
INSERT: (CL=a)·(ST=f) / Clear
ACTIVE
INSERT: (CL=a)·(ST=a) / Start
INSERT: (CL=t)·(ST≠f) / Start
INSERT: (CL=a)·(ST=t) / Start
INSERT: (CL=f)

Fig. 2. State transitions of a timer.

The following triggers implements the rule specified in Example 1:

```
CREATE TRIGGER Delayed_overheat
    TIMER
    ON furnace
    SET INTERVAL '10' MINUTE
    START ON INSERT ( temp > 90 )
    CLEAR ON INSERT ( temp < 80 )
    (Over_heat@FurnaceControl)
```

Following the TIMER and ON clauses, is the timer specification describing the way the timer is controlled. The timer trigger reacts to a specified *start event* (currently only INSERT) by evaluating the start condition and, if it evaluates to true, starting a timer which is active for a specified amount of time. During this time the trigger reacts to *clear events* which may cause the timer to be cleared (i.e. reset and stopped) if the specified *clear condition* is met. If the end of the time span is reached, the system checks whether or not a WHEN condition is declared. If it is, the condition is then calculated, and if the condition evaluates to *true* then the action request is sent.

The three clauses starting with words SET, START and CLEAR constitute the timer definition. The SET clause assigns a time span to the timer (INTERVAL means span in SQL terminology). Included in this definition are also event expressions which control the timer:

- The START clause indicates that the INSERT event starts the timer if the inserted value of "value" exceeds monotonically 90 (or , the previously inserted value one was ≤ 90) . This semantics is possible thanks to the existence of temporal tables in RAPID–the system knows the previously inserted values and it can determine when this condition evaluates to true. Thus, INSERT (or WRITE) is allowed for temporal tables only. In place of INSERT one can use UPDATE which works for all tables. In this case the system calculates the OLD and NEW values of the row, to evaluate the condition.

- The CLEAR clause indicates that any modification command clears the timer when the value descends monotonically below 80. In this case, the new value may be an inserted value, or a modified value. Also here, the INSERT and WRITE sub-events are allowed for history tables only.

Another example shows application of delays to guarding timeliness of change:

Example 2
"If the temperature achieved value 50 it should achieve value 70 in 30 seconds, otherwise increase the heating if the quality is Q."

This case may illustrated in the following way (Fig. 3):

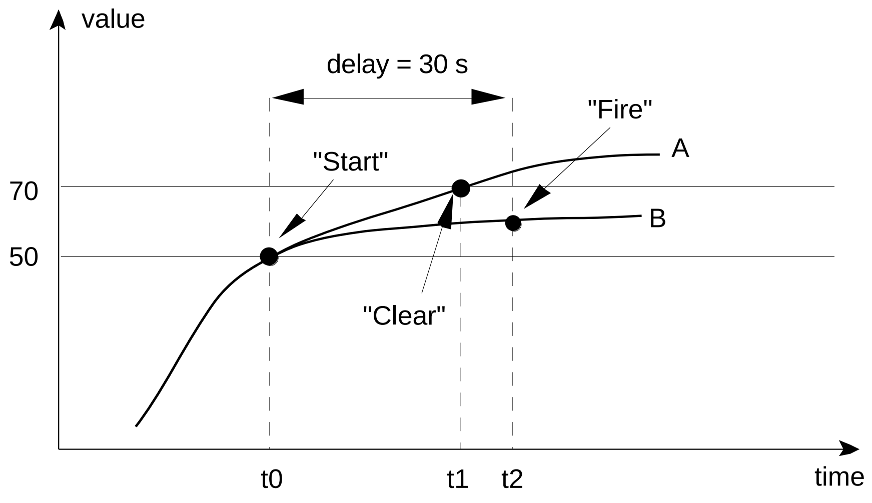

Fig. 3. A timer fires a trigger (case B) when the growth speed is not high enough.

This requires the following trigger to be defined:

```
CREATE TRIGGER Slow_heat_up
    TIMER
    ON furnace
    SET INTERVAL '30' SECOND
    START ON INSERT ( OLD.temp <= 50 and NEW.temp > 50 )
    CLEAR ON INSERT ( temp > 70 )
    WHEN (quality = 'Q')
    (IncreaseHeat@FurnaceControl)
```

Note that the OLD and NEW qualifiers in the timer specification are evaluated immediately after the INSERT command controlling the timer is executed (and not when the trigger is fired, which is the case of the OLD and NEW qualifiers in the WHEN clause).

Any time the controlling event (INSERT) occurs, both the start and clear conditions are evaluated. One reason for this is to ensure that the timer may be started immediately when it is cleared. For example the following trigger will fire if the time span between any two INSERT commands exceeds 5 minutes:

```
CREATE TRIGGER CheckPeriodicInsert
    TIMER
    ON furnace
    SET INTERVAL '5' MINUTE
    START ON INSERT ALWAYS
    CLEAR ON INSERT ALWAYS
    (Notify_break@ActionServer);
```

In this case the start and clear conditions are always true by definition.

Another reason for evaluating always both conditions is the requirement that the timer should not be started if, in addition to the start condition, also the clear condition evaluates to true, as in the case C in Fig. 3.

If the timer need not be cleared at all (unconditional delay), the CLEAR clause may be left out, as in the following example:

```
CREATE TRIGGER Notify
    TIMER
    ON furnace
    SET INTERVAL '10' MINUTE
    START ON INSERT (phase = 'STARTING')
    (Notify_start@ActionServer);
```

4 Counter triggers

Counter triggers react to a certain number of condition-guarded controlling events (currently INSERT events only). The counter specification syntax is the following:

```
counter-specification ::=
    SET number
    INCREMENT ON INSERT (trigger-condition) | ALWAYS }
    [RESET ON INSERT (trigger-condition) ]
    [RESET NOW]
```

For example, one wishes to obtain a notification when four consecutive values are recorded above a certain value level (Fig. 4).

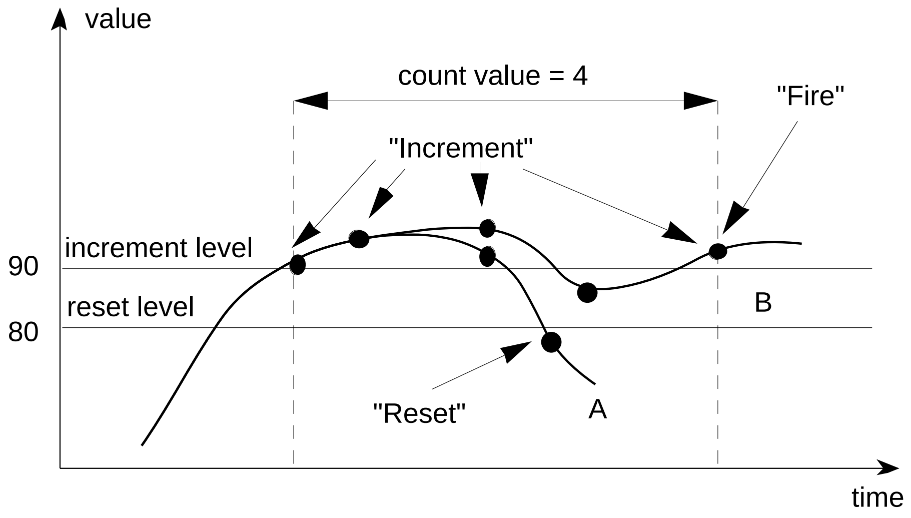

Fig. 4. Counting occurrences of an event.

The functionality shown above may be implemented using the following trigger:

```
CREATE TRIGGER repeated_overflow
    COUNTER
    ON furnace
    SET 4
    INCREMENT ON INSERT (temp > 90)
    RESET ON INSERT (temp < 80)
    (RepeatedUpperAlarm@TempActions)
```

The RESET clause may be omitted, resulting in an unresettable counter which is incremented until it fires. For example, to obtain a notification of each 1000th INSERT, one needs the trigger:

```
CREATE TRIGGER kiloinsert
    COUNTER
    ON furnace
    SET 1000
    INCREMENT ON INSERT ALWAYS
    (Report_kiloinsert@Monitor)
```

The increment and reset conditions may include OLD and NEW qualifiers having the meanings similar to those of the timer trigger. The WHEN condition, if present, is evaluated at the time the trigger is fired. Also the SEND ROW clause refers to the latest row available in the table at that time.

Similarly to timer triggers, both increment and reset conditions are evaluated each time a controlling event occurs. The possible condition values are also the same:

Increment condition	Reset condition	
IN=a		- true by definition (ALWAYS)
IN=t	RS=t	- true by evaluation
IN=f	RS=f	- false by evaluation
	RS=Ø	- empty (RESET omitted)

The corresponding state transition diagram is shown below (Fig. 5):

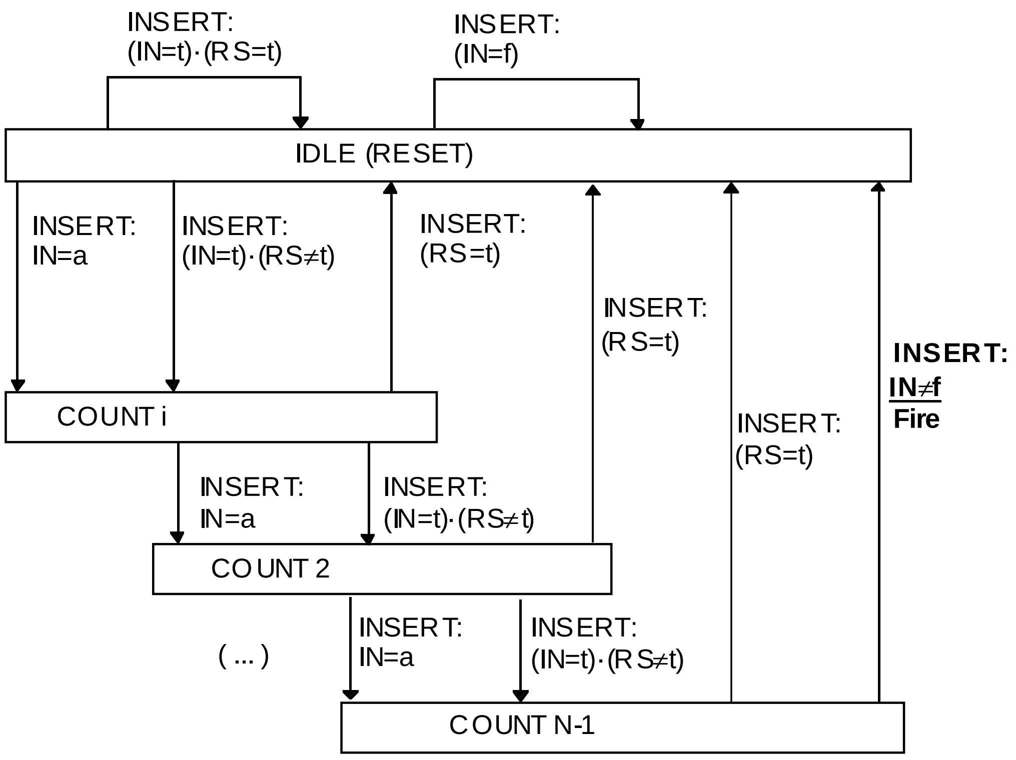

Fig. 5. State transitions of a counter

5 Controlling triggers

Triggers may be removed from the database using the DROP TRIGGER command. For example:

```
DROP TRIGGER trigger_one
```

If a trigger needs to be redefined, the OR REPLACE option may be used, as in the following example:

```
CREATE OR REPLACE TRIGGER Delayed_overheat
    TIMER
    ON furnace
    SET INTERVAL '10' MINUTE
    START ON INSERT ( temp > 90 )
    CLEAR ON INSERT ( temp < 80 )
```

```
        CLEAR NOW
        (Over_heat@FurnaceControl)
```

By default, an active timer is not cleared when the trigger is redefined. The additional CLEAR NOW clause may be used to clear the timer.

Similarly, for counter triggers:

```
    CREATE OR REPLACE TRIGGER repeated_overflow
        COUNTER
        ON furnace
        SET 4
        INCREMENT ON INSERT (temp > 90)
        RESET ON INSERT (temp < 80)
        RESET NOW
        (RepeatedUpperAlarm@TempActions)
```

Here, the RESET NOW clause is used to reset the counter while redefining the trigger. Unless RESET NOW is applied, the redefinition may cause the immediate firing of the counter if the new SET value is less than the current counter value.

All the triggers in a RAPID Server may be switched on or off, using the following commands:

```
    DISABLE ALL TRIGGERS

    ENABLE ALL TRIGGERS
```

The triggers me be also disabled or enabled one by one, for example:

```
    DISABLE TRIGGER Delayed_overheat

    ENABLE TRIGGER Trigger_one
```

6 Implementation notes

In the RAPID system, triggers are implemented as a class hierarchy written in C++. Each trigger class associated with a specific event-type (e.g. INSERT, TIMER, COUNTER, etc.) inherits from the general (abstract) class *trigger*. At any time, it is feasible (for a system developer) to add new trigger subclasses and use method overloading/overriding to shape up a new trigger behavior. Each trigger subclass encapsulates totally the states and transitions of the automaton it models.

7 Summary

We introduced an approach to implementing composite events in active databases as typed automata, and we proposed syntax for timer and counter triggers. The timer triggers incorporate essential history-based event capabilities required in practical active time-sensitive systems, especially in control room applications. The counter triggers enable to detect repetitive events. We also described commands to control execution of triggers in the RAPID system.

References

[CM93] S. Chakravarthy and D. Mishra. Snoop: An Expressive Event Specification Language For Active Databases. Tech. Report UF-CIS-TR-93-007, University of Florida, March 1993.

[Dat94] A. Datta. Research Issues in Databases for ARCS: Active Rapidly Changing Data Systems. SIGMOD Record, Vol. 23, No. 3 (September 1994), pp. 8–13.

[GD94] S. Gatziu and K.R. Dittrich. Detecting Composite Events in Active Database Systems Uhsing Petri Nets. Proc. 4th RIDE Workshop on Active Database Systems. Huoston, Texas, Feb. 14-15, 1994, pp. 2–9.

[GJS92] N.H. Gehani, H.V. Jagadish and O. Shmueli. Composite Event Specification in Active Databases: Model & Implementation. *Proc. VLDB'92 Conf.*, pp. 327-338.

[JP93] J. Jokiniemi and A. Palomäki. Real-Time Databases: A Needs Survey. Research Report No. J-15, Lab. for Information Processing, VTT, Helsinki, February 1993.

[MCD89] Dennis R. McCarthy and Umeshar Dayal. The Architecture Of An Active Data Base Management System. Proc. 1989 ACM SIGMOD Conf. (Portland, Oregon, USA), pp. 215-224.

[Jen+94] C.S. Jensen et al. A Consensus Glossary of Temporal Database Concept. SIGMOD RECORD Vol. 23, No. 1 (March 1994), pp. 52–64.

[SQL3] Working Draft Database Language SQL3, J. Melton (ed.), August 1994, ANSI X3H2-94-329, ISO DBL:RIO-004.

[WC959] J. Widom and S. Ceri (eds.). Active Database Systems: Triggers and Rules For Advanced Database processing. Morgan Kaufmann Publishers, Inc., 1995.

Appendix: The full syntax of trigger definition in RAPID.

```
create-trigger-statement ::=
        CREATE [OR REPLACE] TRIGGER trigger-name
        event-type
        ON table-name
        [timer-specification | counter-specification]
        [WHEN (trigger-condition)]
        (action-list)
        [SEND ROW]
```

Trigger-name defines a name for the event-condition-action rule. Trigger names are unique within a datastore.

ON table-name defines the table into which the trigger is attached.

The WHEN clause contains the condition which is the condition to be evaluated before an action is activated.

```
event-type ::=
        INSERT
        | {UPDATE [OF column-name ...]}
        | WRITE
        | TIMER
        | COUNTER
```

INSERT defines that the trigger is fired on inserted values on the table called table-name, UPDATE defines that the trigger is fired on updated values, and WRITE defines that the trigger is fired on both inserted and updated values. TIMER defines a timer trigger whose behavior is defined in timer-specification. COUNTER defines a counter trigger whose behavior is defined in counter-specification.

OF column-name defines that the condition is checked only if the column-name column (or columns) is affected in the course of executing an UPDATE. If the column clause is not used, the trigger is fired whenever the table is updated.

```
timer-specification ::=
        SET interval-literal
        START ON INSERT { (trigger-condition) | ALWAYS }
        [CLEAR ON INSERT { (trigger-condition) | ALWAYS } ]
        [CLEAR NOW]

interval-literal ::= INTERVAL 'number' SECOND | MINUTE | HOUR | DAY
```

Timer trigger is a normal trigger expanded with entity called timer. INTERVAL defines the time span that the timer will measure when activated. START ON INSERT defines when the timer is started. If trigger-condition is not specified, the timer will be started on every insert to specified table. Otherwise the timer is started only if trigger-condition is satisfied. After the timer is started, it is in active mode and it does not evaluate START ON INSERT condition any more. CLEAR ON INSERT defines when an *active* timer is stopped. It works the same way as START ON INSERT. If the timer is stopped while it is in active mode, the trigger returns to the idle state and reacts only to events defined in START ON INSERT. If the time span specified runs out and the trigger was not cleared, the action will be launched. If there is a WHEN condition specified, the trigger will not launch an action unless the condition is satisfied. CLEAR NOW is used in connection with the OR REPLACE option to clear the timer while redefining the trigger.

```
counter-specification ::=
        SET number
        INCREMENT ON INSERT (trigger-condition) | ALWAYS }
        [RESET ON INSERT (trigger-condition) ]
        [RESET NOW]
```

SET defines the count value the counter fires at. The counter is incremented each time the condition in INCREMENT is satisfied. The counter is reset (set to zero) when the condition in RESET is satisfied. RESET NOW is used in connection with the OR REPLACE option to reset the counter while redefining the trigger.

The trigger-condition is a slightly modified version of the search condition:

```
trigger-condition ::=
        {comparison-condition [AND comparison-condition]}
        | between-condition
        | null-condition

comparison-condition ::=
        [OLD. | NEW.] search-column-name comparison-operator literal
        | literal comparison-operator [OLD. | NEW.] search-column-name

between-condition ::= [OLD. | NEW.] search-column-name
        BETWEEN literal AND literal

null-condition ::=
        [OLD. | NEW.] search-column-name IS NULL
        | [OLD. | NEW.] search-column-name IS NOT NULL
```

If the [OLD. | NEW.] is not used, the NEW column value is assumed. NEW denotes the row resulting from the triggering command. OLD denotes the image of the row before executing the UPDATE command or, in case of the INSERT command, the chronologically preceding row in the history table.

The actions are specified in the action list:

action-list ::= action-name@action-server-name [, action-name@action-server-name]…

The action-list, includes identifiers of *permanent* trigger actions. The notation action-name@action-server-name defines the name of the action to be called and the name of the corresponding Action Server. Each Action Server submits its name when connecting the a RAPID Server. An action function has a standard parameter list including: the trigger name, table name and the OID of the row the trigger is fired on (the triggering row).

The action-list is valid regardless of the existence of the corresponding Action Servers. If an Action Server is in the list and it is not connected at the time of the trigger execution, the action request is skipped and an error message is generated in the Status Log.

A trigger is fired for each affected row (there is no FOR EACH STATEMENT clause of SQL3). The data passed to action functions is thus row-specific.

SEND ROW causes the contents of the triggering row to be sent, following the action function call. The action function may extract the contents of the row using appropraite programming interface methods.