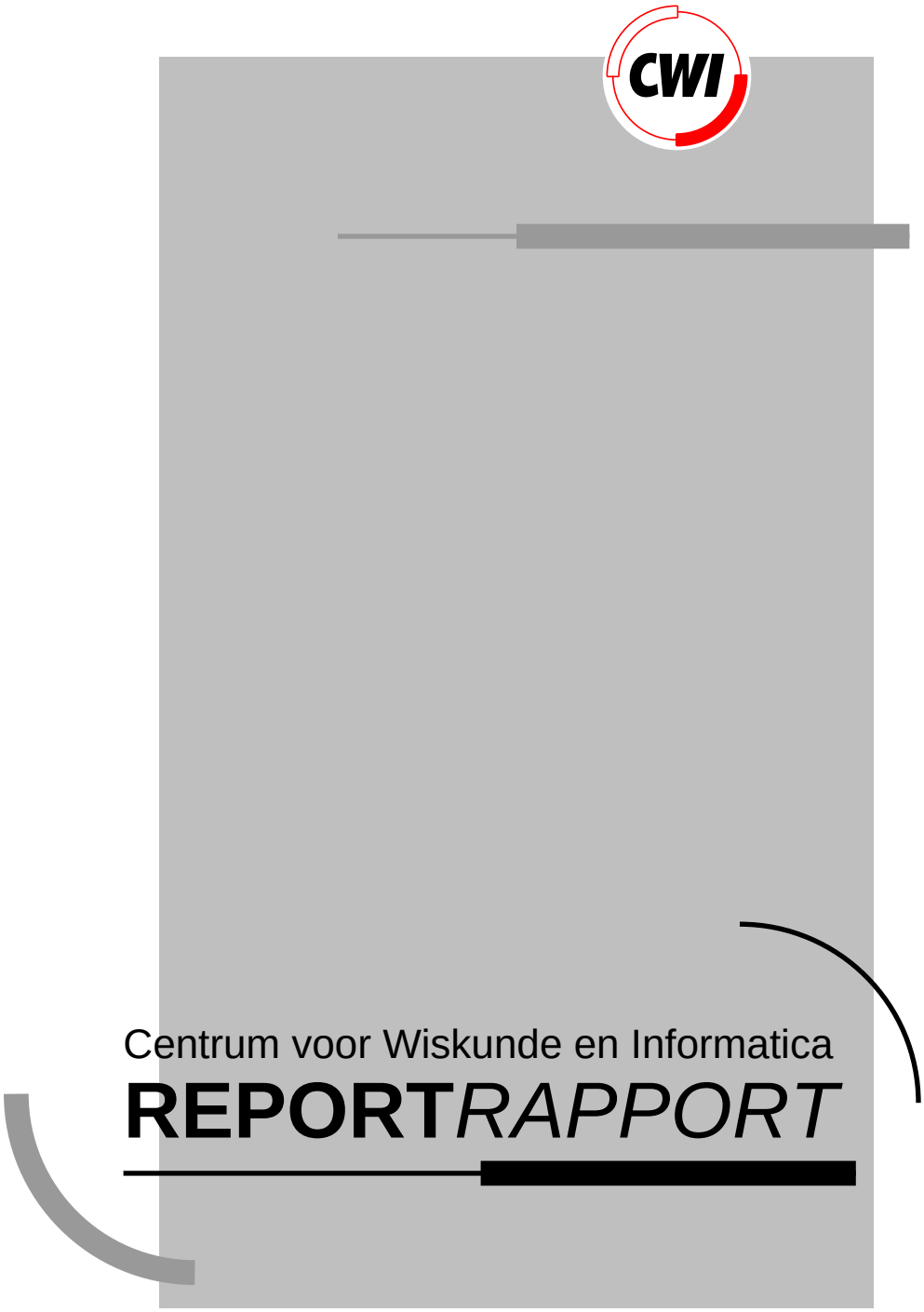

Centrum voor Wiskunde en Informatica

REPORT*RAPPORT*

Towards a design theory for database triggers

A.P.J.M. Siebes, M.H. van der Voort, M.L. Kersten

Computer Science/Department of Algorithmics and Architecture

CS-R9201 1992

Towards a Design Theory for Database Triggers

A.P.J.M. Siebes, M.H. van der Voort and M.L. Kersten
CWI, Kruislaan 413, 1098 SJ Amsterdam
{arno, leonie, mk}@cwi.nl

Abstract

Advances in the area of active database management systems require the development of a trigger design theory, which guides the user in the definition off well-behaving trigger based applications. The development of such a theory requires a formal definition of trigger semantics. This paper describes a framework for such a formalisation of triggers. That is, the parameters of trigger execution and the options for setting them are identified and discussed. Furthermore, the development of a trigger design theory is initiated with the formulation of a sufficient condition for trigger independence.

Categories and Subject Descriptors: H.2.1[**Information Systems**]: Logical Design- *data models*; H.2.3[**Information Systems**]: Languages- *data description languages(DDL); data manipulation languages(DML)*; H.2.4[**Information Systems**]:Systems- *Transaction processing*

General Terms: Algorithms, Documentation, Management

Additional Key Words and Phrases: Triggers, Active Databases, Rules

1 Introduction

One of the prime challenges to designers of new generation DBMS's is to construct active database systems. Active database systems attempt to provide automatic response to events generated internal or external to the system. That is, a conventional database is a passive datastore, all activities are initiated by applications whereas an active database initiates activities itself, without application intervention. This functionality is provided by triggers, which are also known as rules, demons, actors and assertions [9, 10, 20, 18, 22, 25, 27, 28]. Briefly, triggers are a condition-action pair which automatically execute the action whenever the condition, which specifies events and/or interesting database states, holds.

The interest in active databases is twofold. First, active databases satisfy requirements posed on a database by non-standard application areas like CAD/CAM and CIM systems. Second, active databases offer a unifying mechanism for common database management tasks like integrity enforcement and view handling.

One of the requirements of CAD/CAM applications is a flexible constraint mechanism. Not only because the usual transaction abort upon constraint violation does not suffice for CAD/CAM systems, but also because the constraint evaluation point should be controllable, which conventionally it is not. Consider for example chip design: to prevent interference, wires should be placed at a minimum distance from each other. In passive databases, violation of this distance constraint results in transaction abort. In active databases, however, this violation can be repaired by calling a wire re-arrangement algorithm. Thereby, transforming the inconsistent database state into a correct one.
The benefit of application defined constraint evaluation points is illustrated by a CAD application. As checking the construction of a new building design is time consuming, it is beneficial if the architect specifies when to activate integrity enforcement. Unfortunately, in conventional database systems, the evaluation points are fixed. Active databases however, allow for constraint dependent evaluation points.

The CIM area shows another use for active databases, viz., statistics. The design of a production environment is a highly interactive process where a designer is focussed on finding an optimal solution for product assemblages. Once an initial design is made, experiments are done to find defects and to improve the design. Part of these experiments is the collection of statistical information about system behaviour. Active databases offer the capability for automatic gathering of this information.

Taking the requirements from applications into account, the design of an active database system involves the following issues:

Integration of triggers with usual database activities The examples above indicated that applications require control over constraint evaluation points. This can be achieved by a general mechanism for the integration of triggers with database activities. The mechanism should allow for triggers to be executed after each database update and for triggers to be executed at application command. The Hipac project, as described in [10, 11], introduced several trigger execution modes, which are the parameters of the integration mechanism. More work on this subject for relational systems is described in [7, 26] and for object-oriented systems in [5, 12, 15, 20].

Trigger behaviour The behaviour of isolated triggers is not as simple as presented above. On the contrary, several issues have to be dealt with. They can be categorised into two groups: single trigger semantics and multiple trigger semantics. The former deals with variable binding within the condition and with data passing between its condition and action part. The latter deals with execution scheduling of multiple activated triggers and with trigger interference. The following papers describe trigger

semantics [7, 24, 29]. A formalism for the description of trigger semantics is given in [17, 15]

Trigger design theory The design of a collection of triggers is a complicated matter. When the number of triggers in a system increases, the chance of unintentional trigger interference increases as well. This trigger interference may have unforeseen and unpleasant consequences.

As an example, consider a row of adjacent cells on a chip [13]. Except for the end cells, each cell has two neighbours. A cell must always satisfy the following conditions: first, it should be adjacent to but must not overlap its left neighbour (if any) and second, it should be adjacent to but must not overlap its right neighbour (if any). These conditions must be satisfied when a cell is created and when a cell is moved. When a sequence of cells is moved and, thereby, violate one of these conditions, there are two options for integrity repair. The first resolves the conflict by moving the right conflicting cell and the second resolves the conflict by moving the left conflicting cell. When both are incorporated in the trigger system an endless execution sequence of repair actions may occur. The repair is started by moving the right conflicting cell, thereby introducing a new conflict, which could be solved by moving the left conflicting cell. This results in the original conflict and the repair could be repeated (forever).

To prevent this behaviour and to guide the development of well-behaving applications, a trigger design theory is needed. Although the relevance of a design theory is recognised [8, 11, 14, 20, 26], little research on this subject has been published. Work based on the relational model is described in [8, 24, 29] and work based on the object-oriented model is described in [23].

A pre-requisite for the development of a trigger design theory is a formal description of trigger semantics. Some work on this topic has been done for system based on the relational model. This article initiates the formalisation of triggers based on an object-oriented datamodel by giving a framework for such a formalisation. That is, the parameters of trigger execution and the options for setting them are identified and discussed. Furthermore, the development of a design theory is initiated with the formulation of a sufficient condition for trigger independence.

The structure of this paper is as follows. In the next section, we briefly describe a database class hierarchy for the examples in this paper. Moreover, we give an short description of the abstract datamodel underlying our framework.
In Section 3, we develop an abstract definition of a trigger. This definition does not depend on a particular object-oriented data model or DBPL. The reason for this abstraction is twofold. First, the formal introduction of triggers in a DBPL may require language constructs not foreseen during its design. Hence, an early fixation on a DBPL might hinder the actual development of the theory. Secondly, our main concern is to obtain formal semantics of trigger systems. By following Curien's philosophy of making syntax akin to semantics, the discussion of the semantics is greatly simplified. Furthermore, the

semantics of a single trigger is given.
In Section 4, execution models for trigger collections are discussed. That is, given a database state and a collection of (activated) triggers, what is the next state? Furthermore, the integration of triggers with usual database activities are discussed. Together with the previous section, this results in rather abstract semantics for trigger systems.
In the following section, several topics of a design theory are discussed and the development of such a theory is initiated with the definition of *independent* triggers. Finally, in Section 6, conclusions and directions of future research actions are formulated.

2 Prerequisites and an example database schema

In the first subsection of this section, we give an abstract description of a datamodel and introduce some notational conventions. In the second subsection, we give an example database schema. This schema will serve as the basis for the trigger examples in this paper.

2.1 The abstract datamodel

Although we present an abstract framework for the formalisation of triggers, some requirements on the accompanying datamodel have to be given. For example, even for an abstract description of the semantics of trigger applications it is important whether the database consists of *objects* or of *values*.

More specifically, we assume, along the lines of [21, 19, 4], that a database is a set of *objects* in the usual OODB sense. These objects belong to one or more *classes*. Each class definition consists of a description of the *type* of it's objects, the applicable *methods* and the *constraints* to be satisfied.

Moreover, we assume that the set of types, denoted by $Type$, of the datamodel is partially ordered *without* a least element. This partial ordering is called the subtyping relationship, and is denoted by $\leq$. Furthermore,, we assume that the subtype relationship is defined such that for each expression e in the language, and all types σ and τ, $e : \sigma \wedge \sigma \leq \tau \rightarrow e : \tau$ holds: Where, as usual, $e : \sigma$ denotes that e is of type σ [6].

For convenience, we define the relation *bites* on types. To types byte if they have a common specialisation:

Definition 1: Let $\sigma, \tau \in Type$, σ and τ *bite* iff

$$\exists \upsilon \in Type : (\upsilon \leq \sigma \wedge \upsilon \leq \tau)$$

Finally, we assume that the datamodel has an associated query language, such that each query $Q(x_1, \cdots, x_n)$ can be seen as a predicate (with n free variables) on the database state.

We end this section with some notational conventions. Firstly, in this paper we will use DB, or DB_i, to denote a database state. That is, DB denotes a set of objects. Given a state DB, DB^* denotes the set of all tuples that can be formed over DB. More formally:

Definition 2: Let DB be a database state,

$$DB^* = \{(x_1, \cdots, x_n) | x_i \in DB \wedge n \in I\!N\}$$

The elements of DB^* of length n are denoted by $\vec{x}$ or $(x_1, \cdots, x_n)$.

Secondly, for convenience we define the function *Set* to 'unpack' the sets of object tuples:

Definition 3: The function $set: DB^* \rightarrow \mathcal{P}(DB)$ is defined by:

$$set((x_1, \cdots, x_n)) = \{y \mid \exists i \in [1, \cdots, n] : x_i = y\}$$

The function $Set: \mathcal{P}(DB^*) \rightarrow \mathcal{P}(DB)$ is defined by:

$$Set(\{\vec{y_1}, \cdots, \vec{y_n}\}) = \bigcup_{i=1}^{n} set(\vec{y_i})$$

Finally, we will use the abbreviation $Q(DB)$ to denote all object vectors in DB^* that satisfy the query predicate $Q(x_1, \cdots, x_n)$. That is:

Definition 4: Let $Q(x_1, \cdots, x_n)$ be a query predicate and DB a database state,

$$Q(DB) = \{\vec{x} \in DB^* | \, Q(\vec{x})\}$$

2.2 An example database schema

To illustrate the discussion in this paper, we will give example triggers based on the database schema shown in Figure 1. Its syntax is purely illustrative, yet based on trends in object-oriented modelling and object-oriented database programming languages such as [2, 21, 19, 4].

The UoD described by the schema in Figure 1 has four *classes*, viz., *board, cell, wire* and *colour-table*. All four class definitions begin with a description of the attributes their objects have. Only the classes *board* and *cell* have *methods* associated with them directly. All classes have, by default, methods for creating, querying and updating objects.

```
class board with extension Board
    attributes
        cells : set of (c:cell, p:position)
        wires : set of (w:wire, lp:list of position)
    methods
        replace-wire(w1, w2 : wire)
        move-left-conflicting-cell(c1:cell, p1;:position,
            c2:cell, p2:position)
        move-right-conflicting-cell(c1:cell, p1:position,
            c2:cell, p2:position)
end

class colour-table with extension Colour-table
    attributes
        function : string
        colour : string
end
    class cell with extension Cell
        attributes
            type-no : string
            pin-no : integer
            height : integer
        methods
            enlarge()
    end

    class wire with extension Wire
        attributes
            function : string
            colour : string
    end
```

Figure 1: Database schema

3 The definition of triggers

A common definition of triggers, along the line of [10, 14, 20, 28] is **On** *event* **If** *condition* **Then** *action*. Briefly, upon occurrence of the *event*, the *action* is executed when the *condition* holds. Clearly, an event is a general condition. A condition may only address the current database state, whereas an event may also address previous database states. Therefore, we simplify the syntactic definition of a trigger, similar to [9, 20], to **If** *condition* **Then** *action* where the generalised condition may address the current and previous database states.

Whenever triggers are used for integrity enforcement, the object vectors that satisfy the condition are of interest to the action. Therefore, all these object vectors should be selected and processed by the action. As this condition based selection is essentially the same as a database query, the condition will be represented by a query: $Q(\vec{x})$. Furthermore, the **If Then** concept closely resembles the *linear implication* ($\multimap$) from linear logic. Together with explicit quantification, this leads to the following definition of a trigger:

Definition 5: A trigger is defined syntactically and semantically as follows:

Syntax: A trigger T is defined by a formula over a well formed query expression and action part of the form: $\forall \vec{x} : (Q(\vec{x}) \multimap A(\vec{x}))$

Semantics: The semantics of trigger applications are defined in two ways:

logical: Consider both $Q(\vec{x})$ and $A(\vec{x})$ as predicates, where the latter asserts the state of $\vec{x}$ after execution of A. Then $\forall \vec{x} : (Q(\vec{x}) \multimap A(\vec{x}))$ denotes a linear logic formula that states that for each $\vec{x}$ in the database we may conclude $A(\vec{x})$ from $Q(\vec{x})$. After this deduction, we may *no longer assume* that $Q(\vec{x})$ holds.

operational: Let $Trigger$ be a collection of Triggers and $State$ the collection of database states. The operational semantics of a single trigger, T, execution on a database state, DB, are given by the function E of type $Trigger \times State \rightarrow State$ as defined by:
$E(T, DB) = (DB \setminus Set(Q(DB))) \cup Set(\{A(\vec{x}) |\ \vec{x} \in Q(DB)\})$

We will often denote a trigger T as $T = Q \multimap A$, especially if the shared variables are immaterial in the discussion. The following example introduces the trigger syntax used in the sequel of this paper.

Example 1: To prevent interference, wires placed at a board should have a minimum distance from each other. The *minimum-distance* trigger checks this. In case of violation, the wires are replaced in order to obtain a valid wiring.

trigger *minimum-distance*
forall b **in** Boards, e1, e2 **in** b.wires **where**
distance(e1.lp,e2.lp) < min-distance
—∘ replace-wire(b, e1.w, e2.w)

4 Finite collections of triggers

In the previous section, we have defined triggers and their semantics iff there is only one (activated) trigger in the database. In this section, we study the more general problem of several triggers activated at the same time, which is basically a problem of interference.

Given a collection of triggers, the intended global semantics are as follows: For a given database state, determine the set of activated triggers. Choose one to execute and execute it which results in a new database state. Repeat this until there are no activated triggers any more[1].

To illustrate the interference problem, let $T_1 = Q_1 \multimap A_1$ and $T_2 = Q_2 \multimap A_2$ be two triggers, and DB_1 a database state, such that both $Q_1(DB_1) \neq \emptyset$ and $Q_2(DB_1) \neq \emptyset$. Moreover, Let $DB_2 = E(T_1, DB_1)$ and $DB_3 = E(T_2, DB_1)$. There is no guarantee that T_2 is still activated in DB_2, nor that T_1 is still activated in DB_3 as the execution of $T_1(T_2)$ might change objects initially subject to $T_2(T_1)$. Moreover, even if they are both activated, there is no guarantee that $E(T_2, DB_2) = E(T_1, DB_3)$.

Hence, their combined behavior can only be predicted if their execution order is known in advance. In other words, if there is a fixed scheduling (trigger selection) mechanism. That is, if we have a *priority function* [1, 16] for trigger execution, which is studied in the next subsection. Furthermore, the integration of triggers with database activities is discussed in the last subsection.

4.1 Priority functions

In principle there are two ways to define a deterministic priority function:

1. *A state independent priority function*
   That means putting a total order on the set of triggers.
2. *A state dependent priority function*
   That is, for each database state the priority function assigns priorities to triggers. The priority function may use the database contents to determine these priorities.

[1]We do not consider parallel executions in this paper.

Some examples of priority functions are:

1. by *precondition*[2], i.e.: $Q_1(DB) \subset Q_2(DB) \rightarrow T_1 \preceq_{DB} T_2$
2. or inversely: $Q_1(DB) \subset Q_2(DB) \rightarrow T_2 \preceq_{DB} T_1$
3. by inheritance relation between types of selected objects: $Q_1(DB) : \sigma_1 \ \wedge \ Q_2(DB) : \sigma_2 \ \wedge \ \sigma_1 \leq \sigma_2 \rightarrow T_1 \preceq_{DB} T_2$

The above given examples of priority functions are quite simple. More complex ones also take previous assigned priorities in consideration. That is, the priority function possesses a memory. An example (in process algebra terminology [3]) is:

4 by previous priorities, i.e.: $T_1; (T_2 + T_3; T_1)^*$
Note that this might imply that although two triggers are potentially active (e.g. T_2 and T_3) none may be executed as T_1 has precedence.

The advantage of state independent over state dependent priority functions is that the trigger choice does not depend on the actual database state. This simplifies analysis concerning priority functions. However, as the activation of triggers still depends on the actual database state, this advantage is only marginal. Therefore, as state dependent functions are the most general, we define a state dependent priority function. It should be noticed that this definition allows only for priority functions which assign priorities to *activated* triggers. Thereby, excluding priority functions as exemplified in example four.

Definition 6: Let $Trigger$ be the collection of triggers and $State$ the collection of database states.

1. The function $Act_{Trigger} : State \rightarrow \mathcal{P}(Trigger)$ assigns to each state the set of triggers that is activated in that state:
   $Act_{Trigger}(DB) = \{T \in Trigger | \ Q_T(DB) \neq \emptyset\}$

2. A *priority function* is a *partial* function $P_{Trigger} : State \rightarrow Trigger$, such that:
   (a) $dom(P_{Trigger}) = \{DB \in State | \ Act_{Trigger}(DB) \neq \emptyset\}$
   (b) $\forall DB \in dom(P_{Trigger}) : \ P_{Trigger}(DB) \in Act_{Trigger}(DB)$

In the introduction of this section, we sketched the semantics of a (finite) collection of triggers over a database schema. We can now formalise this as follows:

[2]Note, that in this item, as in the next there are some slight technicalities if $Q_1 = Q_2$ or if $Q_1(DB) \cap Q_2(DB) = \emptyset$.

Definition 7: Let *Trigger* be the collection of triggers, *State* the collection of database states and Ps a priority function as defined above. For a state DB_i we have:

logically: we consider DB_i as a predicate giving the state of all of its objects, so we have:
$DB_i \multimap DB_{i+1} = E(P_{Trigger}(DB_i), DB_i)$
That is, from the database state DB_i, we may infer the next database state as $E(P_{Trigger}(DB_i), DB_i)$ (that is, the state that results when we execute $P_{Trigger}(DB_i)$ on DB_i, but after the derivation, DB_i no longer holds.

operationally: we execute $P_{Trigger}(DB_i)$, i.e.
$DB_{i+1} = E(P_{Trigger}(DB_i), DB_i)$

Note that in both cases we can infer new database states until there are no longer active triggers.

4.2 Integration

In this subsection, the formalisation of the integration mechanism, which describes the relation between triggers and usual database activities, is discussed. To formalise the integration mechanism, it seems easiest to formalise database activities as triggers which are activated by external causes (i.e. the user). The integration mechanism then, specifies the relation between triggers, viz. external and internal activated triggers, which is done with a priority function as defined in section 4. A difficulty arises with the existence of transactions. As transactions may be spread over several state changes (e.g. user actions followed by integrity enforcement triggers). This is best attacked by defining the notion of *nested triggers*:

Definition 8: The collection of nested triggers $\mathcal{NT}$ is defined inductively by:

1. if T is a trigger, then $T \in \mathcal{NT}$;
2. if Q is a query, A an action and $T_1, \cdots, T_n \in \mathcal{NT}$, then $(T_1; \cdots; T_n; Q \multimap A) \in \mathcal{NT}$

The intended semantics are defined by:

$$E(T_1; \cdots; T_n; Q \multimap A, DB) = E(A \multimap Q, E(T_n, E(\cdots E(T_1, DB) \cdots)))$$

So for nested triggers, subtrigger T_i should activate subtrigger T_{i+1}; in fact, in general $Q_i \neq \emptyset$ for $i > 1$. The intuition behind the semantics is as follows. T_1 makes a copy of the relevant portion of the database. The following subtriggers work on this copy (i.e. independent of the global state). If all subtriggers execute correct, the result of the nested

trigger is committed. Note, none of the subtriggers changes the global state, hence none of the other triggers in the system is activated until the nested trigger commits.

With this definition, a transaction can be seen as a nested trigger. Each state change of the transaction is represented by a subtrigger of the nested trigger. A benefit of this approach is that it enables a weaker notion of transaction interaction than just serialisability. For, serialisation is but one scheduling mechanism. However, a detailed discussion of nested triggers is outside the scope of this paper.

5 Design theory

The design of a well-behaved collection of triggers is a complicated matter. When the number of triggers in a system increases, the chance of unintentional trigger interference increases as well. Trigger interference may have unforeseen and unpleasant consequences. In order to detect such behavior and to guide the development of well-behaved applications, a trigger design theory is needed. Topics of a design theory are:

fairness An question, raised by the priority assignment is whether all triggers can be executed. That is, given two activated triggers T_1 and T_2 and a priority rule P, are they both executed or is always T_1 and never T_2 executed? This is a fairness problem; which is a difficult theoretical problem. For example, let T_1 have priority over T_2, it depends on the actual database state and the precise *action semantics* of T_1 whether T_2 can be executed or not.

livelock A forever repeating execution of a trigger sequence is a livelock. Usually, such an execution is caused by mutually activating triggers In order to avoid non-terminating applications such trigger combinations should be detected.

independence In the section 4.1, we have introduced the notion of a priority function and we have defined what the semantics of trigger application are with respect to such a priority function. A special priority function is *random choice*. More specifically, it is interesting whether a random choice between the activated triggers does not influence the predictability of the trigger system.

In the following subsection, a sufficient condition for trigger independence is formulated.

5.1 Independence

We start by introducing the notion of independence. Two triggers are independent if their order of execution is immaterial. Clearly, this notion depends on the actual database state,

that is, triggers T_1 and T_2 might be independent with respect to a database state DB_1, while they are dependent with respect to a database state DB_2.

Definition 9: Let T_1 and T_2 be two triggers and DB a database state. Independence of T_1 and T_2 with respect to DB, denoted by $T_1 \perp_{DB} T_2$, is defined by:

$$T_1 \perp_{DB} T_2 \Leftrightarrow E(T_1, E(T_2, DB)) = E(T_2, E(T_1, DB))$$

To analyse this notion of independence, we give a sufficient condition for independence. For simplicity, we assume the action to be query free. This means, the action does not query the database but only accesses objects from the selected object vector.
Furthermore, we introduce the following notation:

Definition 10: Let $T = \forall \vec{x} : (Q(\vec{x}) \multimap A(\vec{x}))$ be a trigger and DB a database state. The write set of T with respect to DB is the set:

$$W(DB) = Set(\{A(\vec{x}) |\ \vec{x} \in Q(DB)\})$$

The read set of T with respect to DB is the set:

$$R(DB) = Set(Q(DB))$$

Thus, the result of the application of trigger T on DB is given by: $E(T, DB) = (DB \setminus R(DB)) \cup W(DB)$ Consider two triggers $T_1 = Q_1 \multimap A_1$ and $T_2 = Q_2 \multimap A_2$ which are both activated on state DB. Define $DB_1 = E(T_1, DB)$ and $DB_2 = E(T_2, DB)$. The triggers are independent if $E(T_2, DB_1) = E(T_1, DB_2)$. Now, assume the following four equations:

1. $Q_1(DB) = Q_1(DB_2)$
2. $Q_2(DB) = Q_2(DB_1)$
3. $Q_2(DB_1) \cap W_1(DB) = \emptyset$
4. $Q_1(DB_2) \cap W_2(DB) = \emptyset$

Then we have:

$$\begin{aligned} E(T_2, DB_1) &= (((DB \setminus R_1(DB)) \cup W_1(DB)) \setminus R_2(DB_1)) \cup W_2(DB_1) \\ &= (((DB \setminus R_1(DB)) \cup W_1(DB)) \setminus R_2(DB)) \cup W_2(DB) \\ &= ((DB \setminus R_1(DB)) \setminus R_2(DB)) \cup (W_1(DB) \setminus R_2(DB)) \cup W_2(DB) \\ &= (DB \setminus (R_1(DB) \cup R_2(DB)) \cup W_1(DB) \cup W_2(DB) \\ &= E(T_1, DB_2) \end{aligned}$$

So, T_1 and T_2 are independent with respect to DB if:

Lemma 1: Let T_1 and T_2 be triggers and DB a database state, such that the conditions (1-4) made above hold, then $T_1 \perp_{DB} T_2$.
Proof: See above.

Clearly, this lemma is not conclusive, that is, $T_1 \perp_{DB} T_2$ might hold while the condition of the theorem is violated. Moreover, the two trigger execution orders still have to be executed before independence can be concluded. However, it allows us to develop a static check for static independence.

First we define static independence as follows:

Definition 11:
Let T_1 and T_2 be triggers and DB a database state variable. T_1 and T_2 are called *statically independent*, denoted by $T_1 \perp T_2$, if

$$T_1 \perp T_2 \Leftrightarrow (\forall DB : T_1 \perp_{DB} T_2)$$

From a computational perspective, static independence is not much better than independence with regard to a given database state. In fact, it is worse, as all database states have to be checked. However, by *typing* the trigger, or better by typing its query and its action part, a static check can be given.

The query type is inferred from the variables in its specification. More specifically, for a query Q, the set V_Q is the collection of all variables in Q. Note, that for the trigger $\forall \vec{x} : (Q(\vec{x}) \multimap A(\vec{x}))$, the set V_Q includes at least all variables in $\vec{x}$, and maybe more. The type of Q is then defined as follows:

Definition 12: Let Q be a query, its type τ is defined by:

$$\tau = \{\tau_x |\, x \in V_Q\}$$

where τ_x denotes the type of x in Q.

The set type in the definition above is simply an unlabeled variant type. So, for subtyping we have:

$$\{\tau_1, \cdots, \tau_n\} \leq \{\sigma_1, \cdots, \sigma_m\} \leftarrow \exists f : \{1 \cdots n\} \rightarrow \{1 \cdots m\}\ (injective) : \tau_i \leq \sigma_{f(i)}$$

All other subtyping rules of the datamodel, such as e.g. those in [6], remain in effect.

Essentially, actions can be seen as methods, hence, they are of a functional type defined in the usual way. The type of a trigger is then defined as follows:

Definition 13: Let $T = Q \multimap A$, with $Q : \sigma$ and $A : \upsilon \rightarrow \tau$, with $\upsilon \geq \sigma$, then

$T : \sigma \multimap \upsilon \cup \tau$

Note, υ appears on the right-hand side of the type because of the semantics of a trigger.

The typing yields an abstraction of the read and write sets:

Theorem 1: Let $T_1 : \sigma_1 \multimap \tau_1$ and $T_2 : \sigma_2 \multimap \tau_2$ be triggers, such that both σ_1 and τ_2 and σ_2 and τ_1 do not bite, then $T_1 \perp T_2$
Proof: Let DB be a database state such that both T_1 and T_2 are activated. Execution of T_1 results in DB_1 and execution of T_2 results in DB_2. The *no-bite* requirement ensures that the requirements of lemma 1 are fulfilled. Hence, $T_1 \perp_{DB} T_2$.

Example 2: As the pins of a chip have a fixed size, the number of pins determines a minimum chip size. This is achieved by the *pin-check* trigger.
To facilitate the chip designer, wires can be coloured. Coupling of function with colour is administered in the colour table. The conformance of a wire's colour to its function is checked by the *wire-colour* trigger.
As these triggers read and write different type of objects, they are mutual independent, which can be checked at compile time.

trigger *pin-check*
forall c **in** Cell **where**
2 * c.length / c.pin-no < pin-size
—∘ c.enlarge()

trigger *wire-colour*
forall w **in** Wire ∧ ct **in** Colour-table **where**
w.function = ct.function ∧ w.colour ≠ ct.colour
—∘ w.colour := ct.colour

However, if the *update-board* trigger (example three) is added, the trigger collection is no longer independent. For, the *pin-check* trigger writes objects read by the *update-board* trigger.

Example 3: The *update-board* trigger checks for overlapping cells and resolves conflicts by moving the left conflicting cell.

trigger *update-board*
forall b **in** Board, e1, e2 **in** b.cell **where**
overlap(e1, e2)
—∘ b.move-left-conflicting-cell(e1, e2)

Clearly, typing triggers results only in a crude approximation of the read and write sets of a trigger. Given a specific DBPL better approximations are feasible. Areas for improvement are:

- *Taking projection into account:* If T_1 only inspects the *cells* of *boards* while T_2 is only concerned with their *wires*, T_1 and T_2 are clearly independent.
- *Take selection into account:* If T_1 is only concerned with red *wires* while T_2 is only concerned with green *wires*, they are again independent.

The first item leads to the notion of an *essential type*, while the second (combined with the first) leads to the notion of an *essential class*. The independence of triggers could then be decided on the basis of their essential class. The formal introduction of these notions, together with the analysis of their properties with regard to independence is, however, outside the scope of this paper.

6 Conclusions

In this paper, we have presented a framework for the formalisation of database triggers. That is, the parameters of trigger execution and the options for their setting are identified and discussed. This includes the execution of a single trigger, the execution of multiple triggers (priority functions) and the integration of triggers with transactions (nested triggers).

This formalisation framework is a step towards the development of a firmly based trigger design theory. Such a theory is needed to guide the development of well-behaving applications. Especially, applications in the area of CAD/CAM and CIM benefit from a design theory as these applications give rise to complex trigger collections and the trigger functionality can not be dismissed.

Furthermore, we have laid the foundation of the design theory with the formulation of a sufficient condition for trigger independence. The typing of triggers provides a mechanism for the static detection of trigger independencies.

Development of a complete design theory requires solutions to issues not covered yet, such as fairness and livelock. Furthermore, a thorough study on nested triggers and their semantics is needed to fully formalise the integration of triggers with user initiated actions.

References

[1] R. Agrawal, R. Cochrane, and B. Lindsay. On maintaining priorities an a production rule system. In *Proceedings of the 3th International Workshop on DBPL*, pages 479–487, 1991.

[2] A. Albano, L. Cardelli, and R. Orsini. Galileo: a strongly typed, interactive conceptual language. In *The ACM transactions on database systems*, volume 10, 1985.

[3] J.C.M. Baeten and P. Weyland. *Process algebra*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1991.

[4] H. Balsters, R.A. de By, and R. Zicari. Sets and constraints in an object-oriented data model. In *Technical Report INF90-75 University Twente, The Netherlands*, 1990.

[5] C. Beeri and T. Milo. A model for active object oriented database. In *Proceedings of the 17th International Conference on VLDB*, pages 337–349, 1991.

[6] L. Cardelli. A semantics of multiple inheritance. In *Semantics of datatypes, LNCS 173*, 1984.

[7] S. Ceri and J. Widom. Deriving production rules for constraint maintenance. In *Proceedings of the 16th VLDB conference*, pages 566–577, 1990.

[8] S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In *Proceedings of the 17th International Conference on VLDB*, pages 577–589, 1991.

[9] D. Cohen. Compiling complex database transition triggers. In *SIGMOD RECORD*, volume 18, pages 225–234, 1989.

[10] U. Dayal, B. Blaustein, A. Buchmann, U.Chakravarthy, M. Hsu, R. Ledin, D.R. McCarthy, A. Rosenthal, S. Sarin, M.J. Carey, M. Livny, and R. Jauhari. The hipac project: Combining active databases and timing constraints. In *SIGMOD RECORD*, volume 17, pages 51–71, 1988.

[11] U. Dayal, A. Buchmann, and D.R. McCarthy. Rules are object too: a knowledge model for an active object oriented dbms. In *Proceedings of the Second International Workshop on Object-Oriented Database Systems*, pages 129–143, 1988.

[12] O. Diaz, N. Paton, and P. Gray. Rule management in object oriented databases a uniform approach. In *Proceedings of the 17th International Conference on VLDB*, pages 317–326, 1991.

[13] N. Gehani and H.V. Jagadish. Ode as an active database: Constraints and triggers. In *Proceedings of the 17th International Conference on VLDB*, pages 327–336, 1991.

[14] E.N. Hanson. An initial report on the design of ariel: A dbms with an integrated production rule system. In *SIGMOD RECORD*, volume 18, pages 12–19, 1989.

[15] R. Hull and D. Jacobs. Language constructs for programming active databases. In *Proceedings of the 17th International Conference on VLDB*, pages 455–467, 1991.

[16] Y.E. Ioannidis and T.K. Sellis. Conflict resolution of rules assigning values to virtual attributes. In *SIGMOD RECORD*, volume 18, pages 205–214, 1989.

[17] D. Jacobs and R. Hull. Database programming with delayed updates. In *Proceedings of the 3th International Workshop on DBPL*, pages 359–371, 1991.

[18] K.M. Kahn and V.A. Saraswat. Actors a special case of concurrent constraint programming. In *OOPSLA 90*, 1990.

[19] M.L. Kersten. Goblin a dbpl designed for advanced database applications. In *DEXA 91*, 1991.

[20] A.M. Kotz, K.R. Dittrich, and J.A. Mulle. Supporting semantics rules by a generalized event/trigger mechanism. In *EDBT 90*, pages 76–91, 1990.

[21] C. Lecluse and P. Richard. The o2 database programming language. In *Proceedings of the 15th VLDB conference*, pages 411–422, 1989.

[22] B. Nixon, L. Chung, D. Lauzon, A. Borgida, J. MYlopoulos, and M. Stanley. Implementation of a compiler for a semantic data model: Experiences with taxis. In *Proceedings of the SIGMOD*, pages 118–131, 1987.

[23] A.P.J.M. Siebes, M.H. van der Voort, and C.J.E. Thieme. Independence. Technical report, CWI technical report, 1992.

[24] E. Simon and C. deMaindreville. Deciding whether a production rule is relational computable. In *ICDT 88*, 1988.

[25] M. Stonebraker, E. Hanson, and C.H. Hong. The design of the postgres rule system. In *Readings in Database Systems*, pages 556–565, 1988.

[26] M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos. On rules procedures, caching and views in database systems. In *Proceedings of the ACM SIGMOD conference*, pages 281–290, 1990.

[27] M.H. van der Voort and M.L. Kersten. Facets of database triggers. Technical report, CWI technical report: CS-R9122, 1991.

[28] J. Widom and S.J. Finkelstein. Set-oriented production rules in relational database systems. In *Proceedings of the ACM Sigmod conference*, pages 259–270, 1990.

[29] Y. Zhou and M. Hsu. A theory for rule triggering systems. In *Advances in Database Technology: EDBT 90*, pages 407–422, 1990.