An Event/Trigger Mechanism to Enforce Complex Consistency Constraints in Design Databases

Klaus R. Dittrich, Angelika M. Kotz, Jutta A. Mülle

Forschungszentrum Informatik
an der Universität Karlsruhe
Haid-und-Neu-Strasse 10-14, D-7500 Karlsruhe, West Germany

ABSTRACT

Data objects in engineering applications, especially computer aided design, show highly complex structures and a lot of intricate dependencies. Hence a large amount of variably composed consistency constraints have to be dealt with. Furthermore, long transactions which are typical for CAD result in the need to tolerate inconsistencies over unpredictably long periods of time. The demands on mechanisms to enforce consistency in design database systems thus differ from those in business and administrative applications. Comprehensive consistency of the design data can only be attained by degrees. The time and extent of checking have to be determined dynamically and under control by the user. In the case of consistency violations, flexible kinds of reaction are necessary. In this paper we propose an event/trigger mechanism to enforce consistency in design databases that complements the transaction-oriented mechanisms suitable for traditional applications. The underlying ideas are derived from exception handling in programming languages. We present in detail the requirements to meet and how our concept copes with them. We also present an implementation that provides reasonable performance.

Categories and Subject Descriptors (Computing Reviews Classification):

H.2.0 [**Database Management**]: General - *Security, integrity and protection*; H.2.3 [**Database Management**]: Languages - *Data manipulation languages (DML)*; J.6 [**Computer Applications**]:Computer-Aided Engineering - *Computer-aided design (CAD)*

General Terms: Design, Languages, Management

Additional Key Words and Phrases: consistency control, design databases, events, triggers.

1. Consistency Requirements in Design Databases

Data objects managed by database systems for design applications - **design database (management) systems (DDBS)** - show highly complex structures and a lot of intricate dependencies. This entails consistency constraints that are by far more numerous and of higher complexity than in traditional business-oriented database applications. In addition to referential integrity, cardinality and existence constraints or key properties, further restrictions resulting from the design environment have to be considered. For example, design data must satisfy predefined standards, design rules evolving from the technology and the methods used or defined by management, and physical laws of the underlying reality. Katz ([KAT83]) mentions three classes of consistency constraints for design applications: **conformity constraints** (the consistency between specification and implementation of a design object), **composition constraints** (correct cooperation of subobjects to form a superobject), and **equivalence constraints** (parts of different representations of a design object have to satisfy given equivalence relations). Of course, **environmental constraints** (certain values within objects meet restrictions to correctly reflect real-world facts) as known from traditional applications are also of concern.

Besides the large number and high complexity of consistency constraints, design environments pose a further consistency problem: **global consistency** in a true sense can only be achieved at the end of the entire design process. Until then, a number of intermediate stages will occur in which subsets of the design data show already **local consistency**. In a hierarchical fashion, the design process should lead from 'lower' to more comprehensive units of local consistency, eventually resulting in global consistency. Note that even local consistency on low levels may be violated for considerable periods of time in a DDBS, as typical design processes follow the method of 'trial and error'. Thus long-term inconsistencies on all levels have to be tolerated without uncontrolled loss of a degree of consistency once achieved.

The time when a consistency check has to take place, i.e. when a certain degree of consistency ist expected, can usually not be detected automatically, nor can checks be tied to database operations in a general way. Frequently the user himself determines that moment, either explicitly or by reaching a certain stage in his design (e.g. the end of a session of graphical editing).

Reactions to the violation of consistency constraints have to be flexible and under user control, too. It is not acceptable to destroy the results of extensive design work by rolling back the database to a checkpoint set way in the past. In some cases, notifying the user will be sufficient; he can then decide himself on what to do to establish consistency. Otherwise, means should exist to execute arbitrary procedures - not only sequences of database operations - as a reaction.

Summing up, a consistency concept for design databases must consider

- to what extent consistency has to be checked,
- at what time consistency has to be checked, and
- how to react to consistency violations.

In contrast to traditional database systems, solutions to **neither** of these parameters can be 'hard-wired' into a DDBS. Thus the interface has to include means to define them at runtime.

Furthermore, the complexity of consistency constraints results in their formulation as logical predicates being only partially feasible. On the other hand, a considerable number of algorithms is available that perform special kinds of elaborate consistency checking (e.g. simulations) for design automation. Any feasible DDBS consistency control mechanism has to be able to make use of these as design companies have invested a lot of money into them.

A rather unusual but characteristic issue for design databases is that a predefined **design procedure** has to be adhered to: only specific design tools should be applied to objects that have reached a certain design stage. Enforcing **design procedure consistency** is one way towards the stepwise provision of global consistency. Finally, engineering design proves to be a rather dynamic area with respect (but not restricted) to consistency. Constraints change more frequently than in traditional applications and vary with technology, project, and even with the particular design team.

Despite or even due to these tough requirements, the advantages of having all consistency control done under the auspices of the DDBS rather than relying on means outside of it are obvious. With the above discussion in mind, appropriate mechanisms have to be suitable for

1. all classes of consistency including design procedure consistency,
2. providing global consistency via various levels of local consistency,
3. formulating consistency constraints as both, a logical predicate or a checking algorithm which yields a boolean result and comprises database accesses as well as higher level programming language constructs,
4. performing consistency checks at arbitrary times; standard times as e.g. termination of a database operation or the classical 'end of transaction' should be optionally usable ('dynamic consistency control'),
5. defining reactions to consistency violations in various ways,
6. defining consistency constraints and reactions to violations at any time ('dynamic definition').

To what extent can concepts developed for conventional database systems fulfill these requirements?

Traditionally, database transactions not only serve as units of synchronization and recovery but also as units of consistency. However, typical transactions in conventional applications are rather short and touch only a small portion of the database. The design environment, in contrast, is characterized by long transactions involving large amounts of data. Some approaches have been proposed to view these long transactions as units of consistency, too. They try to extend traditional mechanisms to cope with requirement 2 above.

One extension is to distinguish between various levels of consistency. [NH82] discuss global consistency and view (classical) transactions as units preserving local consistency of one complete representation of a design object. However, no lower levels of local consistency are supported.

A second major extension are nested transactions ([GRA81], [KIM84], [MOS82]) that can be used to provide an increasing degree of consistency step by step ([EL82], [KE83]). Each transaction is characterized by the set of database ob-

jects it references and by the set of consistency constraints that have to hold before, during, and after the transaction. This results in a number of transaction classes that have to be defined in advance; no further classes can be defined at runtime.

In summary, the classical integration of consistency control into the transaction manager does not cope with all of the problems mentioned above. Especially, requirements 3 to 6 are not met. The proposed extensions address 2, but rather complex structures of nested transactions are necessary. These tend to be difficult to apply and to understand by users and are thus error-prone. Furthermore, it is impossible to accomodate changing consistency constraints once a transaction has been programmed.

A second approach to consistency control in database systems are trigger concepts. They allow to define database operations that conditionally or unconditionally succeed other database operations in an automatic way. This mechanism is particularly suitable to enforce consistency. Parallels to exception handling in programming languages have already been drawn by Eswaran ([ESW76]). He also suggested to offer flexible trigger actions that comprise program control structures in addition to database operations. Furthermore, triggers can readily be defined while the database is in use. However, as actions can only be triggered upon the execution of some database operation, dynamic consistency control at arbitrary times is not yet provided by this concept. Requirement 4 is thus not fulfilled, nor can consistency constraints be specified by appropriate checking algorithms.

To conclude, neither of the discussed concepts can cope with all requirements. However, triggers come much closer to the goal than consistency-preserving transactions. We will therefore extend them to a so-called **event/trigger mechanism** which is based on events that may be raised explicitly at arbitrary times and result in the execution of user-defined actions.

This paper describes the event/trigger mechanism and its application for advanced consistency control by presenting its components and their interaction. A small number or primitives extending a DDBS's data manipulation language is introduced; they are able to deal with a broad range of consistency requirements and avoid a vast set of special-purpose constructs. We give an example of how the concept works and discuss techniques for its implementation.

Note that the concept we propose will by no means replace transactions in a DDBS. They are (probably in an extended form) still necessary for concurrency control and some basic form of recovery. However, events and triggers separate consistency concerns from the rather rigid transaction mechanism and allow enforcement in various ways and at other times within a transaction than at its end only.

There are other areas of database research where similar ideas have been published recently, especially as far as office and process automation are concerned ([FL84], [BP84], [GT83]). In process control, alarm management and time-dependent execution of operations (besides the traditional static consistency control) are emphasized. In office information systems, office procedures have to be executed on a data- and/or time-dependent basis. The solutions devised in the sequel can therefore be of interest beyond the design environment.

2. The Event/Trigger Mechanism

2.1 Overview

The desire to trigger consistency checks at arbitrary times and to execute user-definable reactions to consistency violations suggests to imitate exception handling mechanisms from programming languages and interrupt mechanisms from hardware ([GOO75], [HOR83]). The event/trigger mechanism comprises the following components, and figure 1 gives a first impression of how they are related:

- consistency constraints
- events
- actions
- triggers

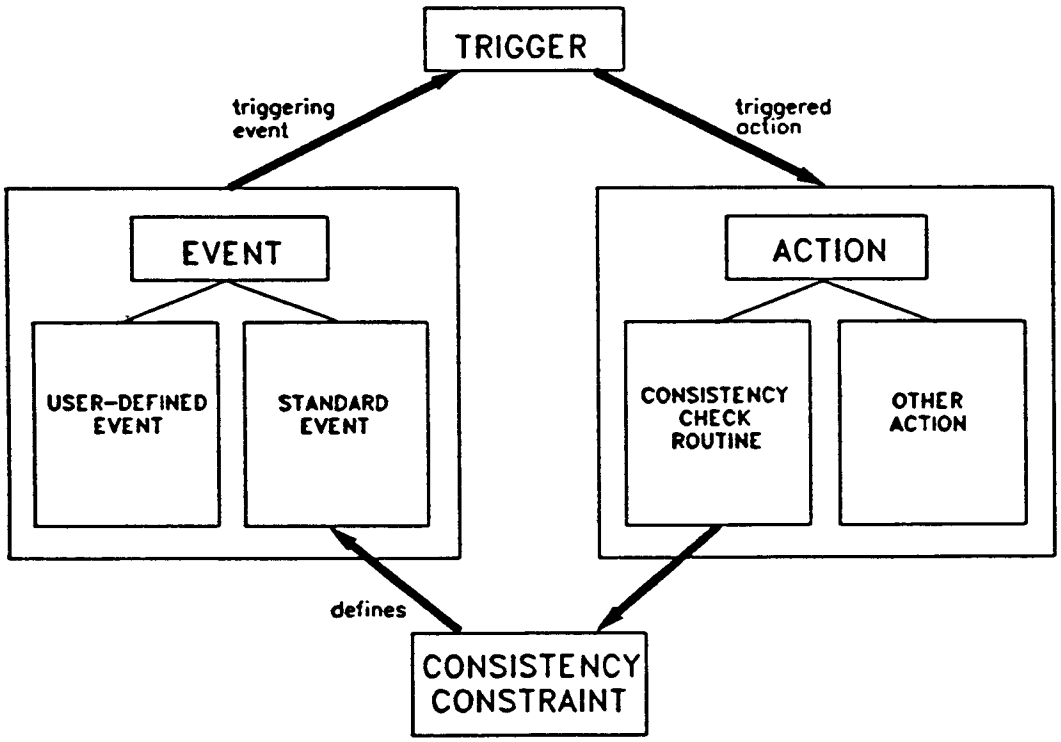

Figure 1

The concept of an event and of the action to be triggered have been borrowed from exception handling. The explicit association of event and action will be installed by so-called triggers comparable to PL/I's ON-conditions. The construct 'consistency constraint' is essential for databases but has no real counterpart in programming languages. Problems concerning scopes of events and exception handlers differ considerably from programming languages and will be dealt with in chapter 2.4.

Definition and manipulation of these constructs involves language support to be provided at the database system interface. Due to the dynamic nature, appropriate statements have to be made part of the DDBS data **manipulation** language. We will subsequently introduce all constructs in detail.

2.2 Components and their primitives

2.2.1 Consistency constraints

Consistency constraints are explicitly inserted into the database by

```
CONSTRAINT <constraint name> = <predicate|boolean procedure>;
```

Defining a consistency constraint of course does not mean to check it immediately. Thus constraints that are not fulfilled when defined can readily be inserted without any consequences. On the right hand-side of the definition, a logical predicate, the name of a stored program to perform the consistency checking, or such a program itself may be specified. Programs may be written in a DDBS host language and include database operations.

The effect of defining a consistency constraint is as follows: if a procedure has been specified, it is checked for syntactical correctness which includes that it yields a boolean result (a specific option for the host language compiler will be necessary to do this). It is then stored in the database in an appropriate format for later execution. If a logical predicate has been specified, it is automatically converted to a checking procedure. This is possible for first order predicates involving database values as only their validity for a **finite** set of database values has to be checked (and not their general validity in the sense of theorem proving).

Consistency control itself, i.e. the execution of the checking procedure, must be initiated explicitly by

```
CHECK <constraint name>;
```

Finally,

```
REMOVE <constraint name>;
```

deletes a consistency constraint from the database. Problems concerning the authorization to define and delete consistency constraints are addressed in chapter 2.4 together with the issue of scopes.

2.2.2 Events

An event is an indicator (represented by a specific identifier) that can be raised to flag a certain situation to the database system. Normally, only the users (including of course all application programs) will know when such a situation occurs; they therefore have to raise the appropriate event explicitly. However, there are some standard situations where the system itself has enough knowledge to raise events implicitly (e.g. when a consistency check yields a negative result).

A number of events will fall into both categories. In these cases it makes sense

that the system automatically only deals with those that are detectable with moderate overhead. It is not reasonable to have a large number of complex conditions checked regularly by the system to figure out whether an event should be raised. Hence we allow for events that are defined and signalled by users as well as for **standard events** that are incorporated into and raised by the system itself. In all cases where the automatic detection of an event is impossible or too expensive, the event has to be raised from outside. This approach is feasible as we can assume that most of the knowledge on what event should be meaningfully raised at what time rests with the user or with the application program (and frequently nowhere else).

The primitives to deal with events are

```
EVENT <event name>;
RAISE <event name>;
ERASE <event name>;
```

Defining an event simply determines its identifier. System-wide uniqueness is established by concatenating user and/or program identification to the given event name. Raising an event causes actions associated with it via triggers (see below) to be executed immediately. If no such reactions exist, RAISE has no effect.

Standard events are implicitly defined (reserved names!) and raised. They have to be both, easily detectable and representing fairly frequent points in design processes (e.g. start and termination of certain database operations). Given a specific database schema, one could also generate type-dependent standard events (e.g. 'insertion of a record of type t'). In particular, the CHECK-operation for constraints as introduced above results in raising either the event <constraint name>.OK or <constraint name>.FAIL which allows for triggering appropriate actions.

As will be seen later, raising an event causes real overhead only when actions are associated with it. However, this would accrue anyway to get the necessary work done. Thus the advantage of having the system control and enforce the timely execution of triggered actions is nearly for free with regard to runtime cost.

2.2.3 Actions

An action is a program to be executed when an event is raised. Again, allowing actions to consist of database operations only would be too restrictive for design applications. Instead, host language **and** DML-facilities have to be available for coding actions.

In contrast to standard events, no standard actions are included in the system. All actions are explicitly inserted, translated and stored in executable format. Action declaration and deletion is done by

```
ACTION <action name> =
   BEGIN
   <sequence of host language/DML-statements>
   END;
DELETE <action name>;
```

Actions may be inserted independently of certain events; the association of actions with events is done by defining triggers (to be introduced next).

2.2.4 Triggers

A trigger is a pair (E: event, A: action) with the meaning that A is to be executed immediately whenever E is raised. Triggers may especially support consistency checking by using a CHECK-operation as their action.

The trigger definition statement is

```
TRIGGER <trigger name> =
   ON <event name>
   DO <action name|action>;
```

with actions being either predefined or specified right in place. Triggers are activated/deactivated by

```
ACTIVATE <trigger name>;
DEACTIVATE <trigger name>;
```

Following its definition a trigger has to be activated to show its desired effect; deactivated triggers do not cause any action should the associated event be raised. Deletion of triggers is by

```
DROP <trigger name>;
```

There are a couple of questions arising with trigger definitions:

1. How should processing of an application program continue after execution of a triggered action?
2. Can multiple triggers per event be defined?
3. How are nested triggers handled, i.e. triggers whose actions in turn raise events that are part of defined triggers? Recursive triggers are a special case of this situation.

As of 1), processing after execution of a triggered action can optionally continue in one of two ways. First, one can go on to execute right after the statement that caused the event to be raised ('resume', comparable to procedure calls). Otherwise, immediate termination of the complete operation that raised the event may be demanded ('exit'). In this case, further research has to investigate how those operations should be defined exactly and how they can be forced to terminate.

Concerning the second question, it is desirable to be able to execute an ordered or unordered sequence of actions when a single event has been raised. Consider for example an event 'end of graphical editing of a design object' that is supposed to trigger a number of consistency checks. Using a priority scheme allows for defining the desired sequence. We therefore allow multiple trigger definitions to refer to the same event. However, this has to be done carefully as interactions within the involved actions may render undesired results. Serializability of unordered sequences remains to be investigated. Multiple definitions

will have to be prohibited where serialization would cause difficulties.

Let us look at question 3). Hierachical and recursive data structures are rather frequent in the design environment. Nested triggers are well-suited to deal with them ([ESW76]), hence a DDBS should support them. Two problems occur with nested triggers, namely how to recognize them and how to guarantee termination. When defining triggers, nesting can not generally be detected as this would involve analysis of all trigger actions. Hence, recognition has to be left to run-time which is also true as far as termination is concerned. A maximal depth of nesting or a time limit have to be imposed whose expiration will undo the effects of prior actions of the trigger in question. Nested trigger actions therefore have to be atomic to provide for complete recovery (which has to hold for every single trigger action anyway).

As a conclusion, multiple trigger definitions and nested triggers should remain under tight control and permitted for privileged users only. General usability of this concept should assume that the associated set of actions can be restrained such that undesired interactions are impossible (e.g. when all actions involved consist of message printing only). Further investigations are necessary to give criteria for selecting 'undangerous' sets of actions.

2.2.5 Summary

Let us summarize all previously defined language constructs of the DDBS-interface:

a) **consistency constraints**

definition:
```
CONSTRAINT <cc-name> = <predicate|boolean procedure>;
```
checking:
```
CHECK <cc-name>;
```
deletion:
```
REMOVE <cc-name>;
```

b) **events**

definition:
```
EVENT <e-name>;
```
release:
```
RAISE <e-name>;
```
deletion:
```
ERASE <e-name>;
```

c) **actions**

definition:
```
ACTION <a-name> = <program>;
```
deletion:
```
DELETE <a-name>;
```

d) **triggers**

definition:
```
TRIGGER <t-name> =  ON <e-name> DO <a-name|action>;
```
activation:
```
ACTIVATE <t-name>;
```

```
deactivation:
   DEACTIVATE <t-name>;
deletion:
   DROP <t-name>;
```

It is worth mentioning that consistency checking is only a special application of the event/trigger concept; the checking of consistency constraints is a special kind of action, and events, triggers and (general) actions may be used for whatever is desired in a DDBS.

2.3 Example

The following example is borrowed from the area of electronic circuit design. Before starting editing a layout, we want to check whether the logical design has been completed and tested by simulation (i.e. design procedure consistency is enforced). We assume that information about the current design state is available from the database system (by using the predicates logic_defined, logic_simulated). During layout editing, the design rule 'width of wires will never be less then 2 micrometers' has to be enforced. At the end of layout design the result has to be compared to the logic design.

```
CONSTRAINT k1 = logic_defined AND logic_simulated;
CONSTRAINT k2 = wire.width >= 2;
CONSTRAINT k3 = BEGIN <algorithm for logic-layout-comparison> END;

EVENT layout_edit_start;
EVENT layout_edit_end;

ACTION a1 = BEGIN <reject_layout_edit> END;
ACTION a2 = BEGIN <reject_insert> END;
ACTION a3 = BEGIN <signal_consistency_violations> END;

TRIGGER t1 = ON layout_edit_start DO CHECK k1;
TRIGGER t2 = ON STANDARD_insert_wire DO CHECK k2;
TRIGGER t3 = ON layout_edit_end DO CHECK k3;
TRIGGER t4 = ON k1.FAIL DO a1;
TRIGGER t5 = ON k2.FAIL DO a2;
TRIGGER t6 = ON k3.FAIL DO a3;
```

Now if the event layout_edit_start is raised by the graphics editor, k1 will be checked. Should the result be negative, an appropriate reaction according to the program for action a1 is taken. As no triggers have been defined for the event k1.OK, a successful check of k1 will allow the graphics editor to continue right away. The standard event STANDARD_insert_layer is assumed to be raised by the system when a new database element 'wire' is inserted.

2.4 Scopes

Similar to exception handling in programming languages, the scopes of all introduced elements have to be considered for the DDBS event/trigger mechanism. The solutions, however, will be different as no comparable block structure is available. Furthermore, 'scope' has a twofold meaning here, namely the

realm in which elements (i.e. their identifiers) are visible, and the realm of trigger activations. Meaningful scopes can be classified as follows:

- all projects represented in the database
- one specific project
- a team of designers within a project
- one specific designer

This classification is a hierarchy in the sense that consistency constraints, events, actions and triggers defined at a higher level are visible at lower levels, but not vice versa. For a specific designer, for instance, all consistency constraints of his project, his team, and himself are visible and can thus be checked.

According to this hierarchy, triggers can be activated and deactivated locally. Activation statements on lower levels will override those of higher levels for the local environment.

Definition and deletion of all constructs as well as (de)activation of triggers is possible on all levels but will require specific authorization to reflect usual project organization reglementations. Standard events can only be raised by the system, while triggers using standard events have to be defined explicitly on the desired level.

3. Implementation Techniques

For an implementation of the proposed concept, we will give runtime efficiency the highest priority, at the expense of storage efficiency. In a design environment, this can be explained as follows. Typical computer architectures in this application area will consist of a central file or database server and a network of workstations. Increasing (virtual) main memory capacities can be expected for these workstations in the near future. Design engineers usually deal with largely disjoint design objects or parts thereof. All defined elements used in the event/trigger mechanism can be partitioned accordingly. Only those relevant for the current design object and for the current designer have to be stored at his workstation, then. On the other hand, runtime efficiency of all mechanisms is important regarding the complex operations that have to be performed frequently on data structures in design environments.

We use four hash-tables to provide fast access paths to consistency constraints, events, actions and triggers in main memory (of course, all this information is held in the database itself, too, and the hash tables are loaded from there):

- an event table ET,
- an action table AT,
- a consistency constraint table CCT, and
- a trigger table TT,

whose entries can be reached via the respective identifiers (e.g. the event name). Although the event/trigger mechanism would work with only one compound data structure, we use the access paths provided by all these tables for fast insertion and deletion which is important due to the dynamic definition capabilities provided. The event table is of particular significance for raising events, as it is the case for the trigger table for trigger (de)activation. All exe-

cutable modules implementing actions and consistency checking algorithms are stored in an action pool that is referred to directly wherever possible. Figure 2 gives an overview of the resulting table structure that will be detailed in the following.

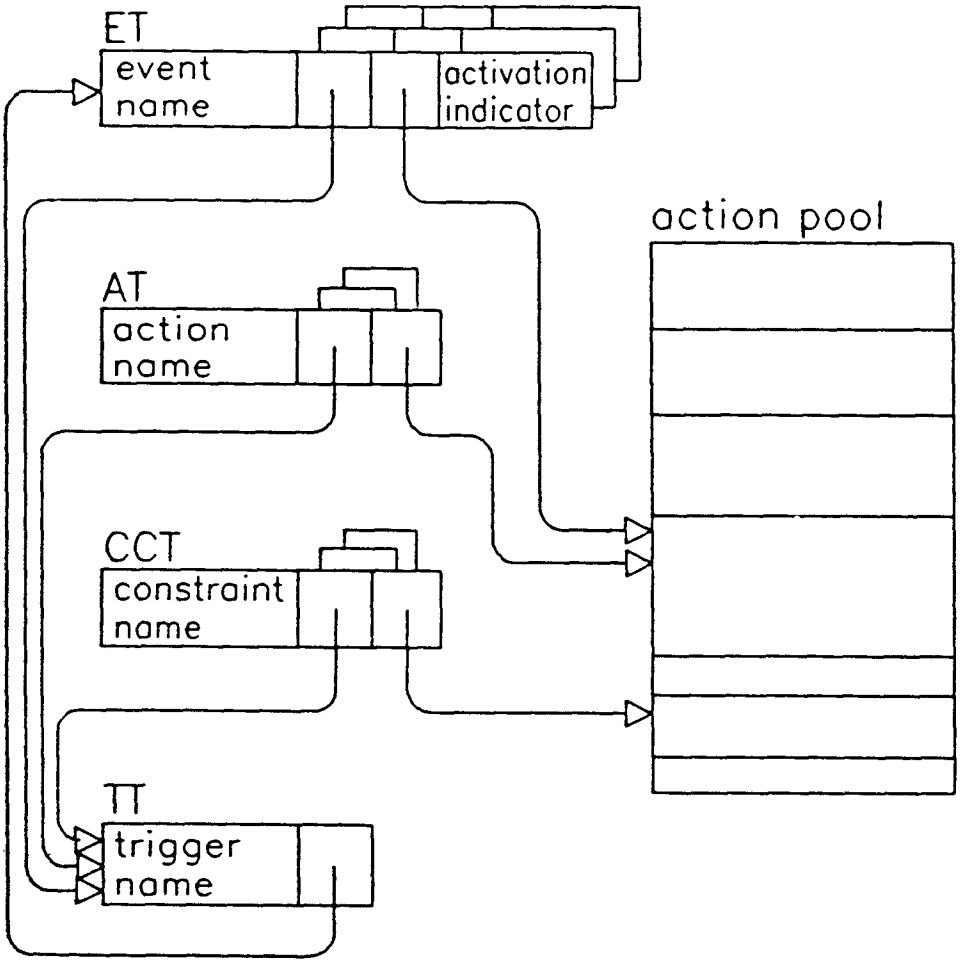

Figure 2

a) **event table** (ET)

All events including standard events are inserted into the event table. Information can be found there concerning which trigger has been defined for a given event, which action has to be executed as a reaction, and whether the trigger is currently activated. Implementation of ET has to allow for variable length entries when multiple triggers per event are permitted.

ET-entries are created as part of the EVENT-operation; references to TT and to the action pool result from trigger definitions specifying the event in question. When an event is erased, its ET-entry and all triggers based on it will be deleted

(while the actions defined with these triggers continue to exist).

The RAISE-operation is the most peculiar one of the whole concept. Informally, it works as follows:

```
RAISE <e_name>:    IF <e_name> is in ET
                   THEN FOR every trigger associated with <e_name>
                        DO IF trigger is activated
                           THEN execute associated action
```

Note that all necessary information can be reached directly from the event's ET-entry. Also, RAISE is very fast when no triggers exist or when they are inactive.

b) **action table** (AT)

For every action, references to all triggers containing it are stored together with a reference to its executable module. Action table and action pool are kept seperate due to runtime efficiency as the action pool can usually not be stored in main memory. AT-entries are created when actions are defined; trigger definitions will add the necessary references. Also, deletions of AT-entries result in the deletion of triggers based on these actions.

c) **consistency constraint table** (CCT)

For every consistency constraint, CCT keeps entries for all triggers that contain its checking as the only associated action. Thus, CCT is a special action table for CHECK-actions.

d) **trigger table** (TT)

To provide for efficient access, only the associated event has to be stored for every trigger; further information is already available from ET.

All trigger operations use TT as a starting point. For example, ACTIVATE will retrieve the relevant ET-entry by following the reference found in TT. On trigger deletion, all other tables that refer to TT have to be updated; however, no entries have to be deleted there.

Finally, we have to deal with the question of how to store and activate action and consistency checking procedures in the action pool. Two possibilities can be considered:

First, actions might be stored as linkage modules. However, many operating systems lack features for dynamic linking, at least for runtime-created modules. As an alternative, actions might be stored in executable form and called when needed. Communicating data to these modules can be done by parameter passing and/or interprocess communication. The action module in turn may cause accesses to the database. Using the UNIX operating system, we proceed as follows: The action pool consists of a tree of files that may be structured according to projects, design teams etc. Every action corresponds to one executable file. As soon as an action is raised, a subprocess is created whose

task is to trigger execution of the associated file. Information exchange between process and subprocess uses an interprocess channel (UNIX pipe).

4. Conclusions

We have introduced an event/trigger concept that is suitable to support the complex consistency control problems in design databases. Moreover, it can be used for any action that needs to be triggered and is not restricted to pure consistency control. We have shown that a small set of basic DML-constructs will do to fulfill various requirements, even some that may not be known to date. Implementation of these concepts can be accomplished with reasonable overhead.

Although primarily intended for use in design environments (classical CAD as well as software engineering environments), the mechanisms may likewise be appropriate for other engineering applications or for office information systems.

Some problems remain to be investigated in future research:

- How can our approach be integrated with efforts to include dynamic capabilities into database systems (cf. [MC83a], [MC83b])? Especially in the design environment, virtual database values have to be computed by user-defined functions. Similar requirements to those found with triggered actions for consistency control exist.
- Can the event/trigger concept be used for further purposes, e.g. for design data recovery? Long design transactions are unsuitable for recovery. It would be desirable to have recovery actions triggered by users or automatically from time to time.
- Problems encountered with multiple and nested trigger definitions have to be further studied. We would like to find out what controls concerning the consistency of a set of events, actions and triggers itself could be carried out by merely analyzing the structure of this set.
- Finally, we want to look into the question whether it makes sense to parameterize consistency constraints and actions with database objects, and what consequences this extension would have for our concept.

Bibliography

[BP84] Bracchi, G. and Pernici, B., *SOS: A Conceptual Model for Office Information Systems*, in: Data Base **15** (1984) pp. 11-18.

[EL82] Eastman, C. and Lafue, G., *Semantic Integrity Transactions in Design Databases*, in: *File Structures and Databases for CAD*, J. Encarnacao and F.-L. Krause (eds.), North Holland Publ. Comp. (1982) pp. 45-54.

[ESW76] Eswaran, K. P., *Specifications, Implementations and Interactions of a Trigger Subsystem in an Integrated Database System*, IBM Research Report RJ1820 (Nov. 1976).

[FL84] *Functional Interface of a Database Computer for Process Data Processing (in German)*, Ferkinghoff, B. and Liedtke, R.-P., Computer Science Research Laboratory at the University of Karlsruhe, Research Report 1/84 (May 1984).

[GOO75] Goodenough, J.B., *Exception Handling: Issues and a Proposed Notation*, Comm. ACM **18** (Dec. 1975) pp. 683-696.

[GRA81] Gray, J., *The Transaction Concept: Virtues and Limitations*, Proc. 7th Int. Conf. on VLDB (1981) pp. 144-154.

[GT83] Gibbs, S. and Tsichritzis, D., *Data Modeling Approach for Office Information Systems*, ACM Trans. on Office Automation Systems **1** (1983) pp. 299-319.

[HOR83] Horowitz, E., *Fundamentals of Programming Languages (chapter 9)*, Springer (1983).

[KAT83] Katz, R.H., *Managing the Chip Design Database*, IEEE Computer **16** (Dec. 1983) pp. 26-36.

[KE83] Kutay, A.R. and Eastman, C.M., *Transaction Management in Engineering Databases*, Proc. Database Week, IEEE Computer Society Press (1983) pp. 73-80.

[KIM84] Kim, W. et al., *Nested Transactions for Engineering Design Databases*, Proc. 10th Int. Conf. on VLDB (1984) pp. 355-362.

[LP83] Lorie, R. and Plouffe, W., *Complex Objects and their Use in Design Transactions*, Proc. Database Week, IEEE Computer Society Press (1983) pp. 115-121.

[MC83a] Melkanoff, M.A. and Chen, Q., *An Experimental Database which combines Static and Dynamic Capabilities*, Proc. Database Week, IEEE Computer Society Press (1983) pp. 53-61.

[MC83b] Melkanoff, M.A. and Chen, Q., *Integrating Action Capabilities into Information Databases*, Proc. 2nd Int. Conf. on Databases, Cambridge (Sept. 1983).

[MOS82] Moss, J.E., *Nested Transactions and Reliable Distributed Computing*, Proc. 2nd Symp. on Reliability of Distributed Software and Database Systems (1982) pp. 33-39.

[NH82] Neumann, T. and Hornung, C., *Consistency and Transactions in CAD Databases*, Proc. 8th Int. Conf. on VLDB (1982) pp. 181-188.