A GENERAL MODEL FOR VERSION MANAGEMENT IN DATABASES

Peter Klahold, Gunter Schlageter and Wolfgang Wilkes

University of Hagen, Praktische Informatik I
Postfach 940, D-5800 Hagen, West-Germany

Abstract

In this paper we introduce a general model for version management expressed by the concept of version environments. A version environment offers two mechanisms for structuring the version sets of objects: graphs and partitions. By the use of views, constraints and transactions the version environment may be tailored to specific user requirements. The embedding of application tools into the version environment provides the users with their specific application environment which consists of the objects, their version structures and the tools operating on them. The proposed concept is more general and more powerful than the concepts published so far; it is shown, how well known version concepts can be implemented by means of version environments.

1. Introduction

Many application areas now or in future supported by database systems cannot be modelled in a sufficient way by representing only the present state of the world. Instead it must be possible to model past and future states as well as states existing parallel to each other. These requirements led to the development of various version models tailored to specific application areas. Thereby new concepts were introduced or taken from application areas (for instance time version, alternative, variant, etc), which all describe some facets of a general version concept.

In order to support these different application areas with their specific version concepts by databases, we have to find a common base for these different concepts. We then can attempt to integrate this common base into data models and, possibly, to support a version kernel by lower levels of the dbms.

This work was supported by Deutsche Forschungsgemeinschaft, IBM Germany and BMFT, contract no. NT 2823 4.

In this paper we introduce the concept of a **version environment.** A version environment models complex version structures, which are based on some primitive mechanisms. To motivate these mechanisms, at first we sketch some version concepts known from literature. In chapter 3 we give a survey of our model, which in chapter 4 is described more in detail by means of an example in the area of CAD. In chapter 5 we show, how our model can be extended by an operational component, which defines the operations on the version structures of a version environment. Finally, we model some of the version concepts of chapter 2 in terms of our model.

2. Version Concepts: Some Approaches

We try to develop a uniform version model based on elementary structuring mechanisms extracted from existing version concepts. In this chapter we describe some version concepts known from literature.

2.1 Commercial Applications

In commercial applications old states are ordered in accordance with a time attribute to allow controlling of (commercial) transactions and their results (auditing). However, it turned out that there are several dimensions of 'time' in this environment.

Example: Queries on salary histories

1) What salary did an employee get in month m?
2) What was the salary of an employee for his work done in month m?

To answer query 1, we have to know the date of payment. For query 2 we need the period for which salary was paid; the time of payment is not of interest. A subsequent adjustment of salary will lead to different results of the two queries. /Härd/ distinguishes three aspects of time to represent as much semantics as possible:

- time of realization, i.e. when the occurrence of an event is recognised or a future event is specified.
- time of storage into the database
- time of validity

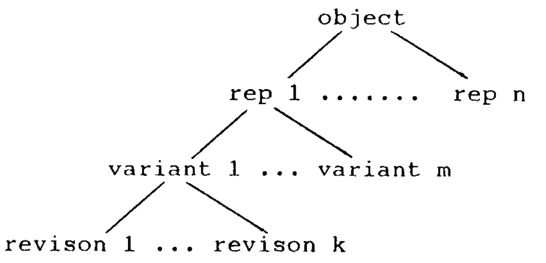
According to the three points of time, there are three different ordering relationships on the set of objects.

2.2 CAD/CAM applications

In CAD/CAM, the notion of version is of great importance. Versions are used for different purposes, e.g. to illustrate the life-cycle of a design object, to document the process of development, or to represent variants in production. This shall be examplified in the following.

/MüSt/ decribes a time version model for CAM databases which allows the versions to be ordered with respect to the beginning of production. For this reason the model offers operations to move the versions along the future part of the time axis.

/Lock/ introduces the following version scheme:



Representations are different views of an object (e.g. logical design and layout of a VLSI circuit). Variants, often called alternatives, are different approaches to a problem, they are developed in the respective revisions.

/KaLe/ introduces a version model defining the life-cycle of objects (and consequently the process of design). Every version adopts a certain state. There are operations

- to transfer a version from one state to another(e.g. ARCHIVE: released version -> archived version)
- to derive a new version v2 in state s2 from an existing version v1 in state s1 (e.g. CREATE ALTERNATIVE: in progress version -> alternative).

This allows to represent the production phases as well as the development stages of an object in one concept. The state of a version indicates its meaning.

A generalisation of these approaches is described in /DiLo/. Instead of the fixed version structures, version clusters are introduced. They allow to group versions at will. Thus, the version structure can be adapted to the application. Clusters are implemented by pointer structures on top of the extended SQL database.

In /KSUW/ the documentation of design history is emphasized. Design rather progresses by trial and error than in a linear way. This leads the authors to the concept of a version graph which has meanwhile been used by other authors, too /KACh/. The version graph allows to represent flexible order relationships by describing parallel alternatives and linear versions in one concept. Version graphs are one element of the general version concept introduced in this paper.

3. A survey of the general version model

In the following we introduce a general version model. At first we sketch our idea of versions and objects. Then we define the basic version structures, which are used to model the applications described above. Finally, we introduce 'version environments' as a means of defining and manipulating version structures.

We start from an intuitive notion of objects and versions. The user thinks in terms of objects where an object constitutes a unit or an entity for him. An object is not static, it develops over time. Therefore the notion of object is (at least) twofold: Sometimes the user considers the total of all states as his object, sometimes he regards a particular state as the unit he is interested in.

A single state of the user's entity is called 'version'. We use the term 'object' when we talk about the set of all versions (or states) of this entity. The actual information is stored in the versions, the object is only a unit which comprises all versions (and what they have in common). This general structure is similar to the version concept of /BaKi/ and /KiBa/.

Versions are, for instance, represented by (sets of) tuples in a database. They comprise the total of information necessary to describe a state of an object, i.e. they can be regarded as updated copies. This does not exclude techniques to save space at lower levels. For reasons of simplicity, objects and versions are identified in this paper by unique identifiers <oid> and <vid>.

In the models sketched in chapter 2, versions are related to each other in different ways. Two main structures can be distinguished:

1. Versions are ordered (not necessarily totally) by various relationships (time relations, development history). To model these orders the general version model offers version graphs.

2. Versions are classified according to specific properties (valid/invalid, in progress/alternative/effective/etc.). The set of versions is partitioned, with every version belonging to a certain class. A class may determine the operations applicable to the versions in it.

In this paper we propose a concept combining these means of structuring versions.

Up to now we distinguish two main levels: on the one hand, the objects as a set of versions, on the other, the relationships between versions. The concept of **version environment** integrates these two levels: A version environment consists of a set of objects and a set of version structures; the version structures are based on the two structuring mechanisms graph and partition. Every object inherits the version structures of its environment. Thus, all objects of an environment have common version structures. Many version environments can be managed by the system; in principle, every object may possess its own version environment.

The model is organized in four levels. The lowest level 1 realizes the notion of versions and objects. Levels 2 and 3 realize the concept of version environments. The fourth level allows to adapt the model to the application environment by importing tools as transactions. Every level offers a functional interface the operations of which may be used by higher levels. The model looks as follows:

```
        +---------------------------+
        ! application environments !
        +---------------------------+

+--------------------------------------+
!        version environments          !
! view, constraint, complex operation  !
+--------------------------------------+

+--------------------------------------+
!              structures              !
! version graph            partition   !
+--------------------------------------+

          +---------------------+
          !      versions       !
          +---------------------+
```

The lowest level (level 1) offers basic operations on versions and objects: Objects may be created and deleted, and there are operations for inserting, updating, deleting and selecting qualified versions of an object. At this level all versions of an object are considered separately; no relations between them are known.

Level 2 manages the version graphs and partitions. There are operations (1) to define and delete version graphs, as well as partitions and their classes, (2) to manipulate the version structures (creating and deleting of edges between versions, shifting versions between classes) and (3) to retrieve versions using the structures (predecessor, successor in version graphs, membership in classes).

Level 3 provides a mechanism to define views adapting the version environments to specific requirements of individual (groups of) users. These views form new (virtual) objects with their own environments. To control integrity, rules may be specified which disallow operations on versions and version structures at certain data states. Moreover, we allow to define complex operations as a sequence of basic operations, e.g. to insert versions into the object and the corresponding structures at one time.

Level 4 expands the version model by an important feature: By embedding application tools into version environments it provides a mechanism for the operational adaptation of the version model to specific application areas. Thus, the system can guarantee the correct access to data.

The version environment is managed by an environment administrator. He defines the version structures of the environments and provides users with specific operations and views. The user works on existing objects with predefined version structures only.

4. Using version environments

In this chapter the most important operations of the general version concept are illustrated by means of an example of the CAD environment. The complete list of operations can be found in the appendix.

We consider the following design environment for chip objects:

1. Chip objects have to supply versions to keep former states of the design.

2. The versions are arranged in a history-graph to retain the development of the object. The edges of the graph have the meaning 'is derived from'.

3. The time of creation of versions must be documented. The versions have to be ordered in a linear way with respect to their time of creation.

4. The versions are grouped into certain consistency classes according to the tests performed successfully (e.g. by simulation programs). The classes are: unchecked, test_a, test_b and consistent.

4.1 Basic version handling

To represent this specific design environment two version graphs and one version partition are used which have to be adapted in correspondence with the requirements. To retain the time of creation, chip objects must have the attribute 'time'.

After defining the version environment 'chip_design' by

```
DEFINE ENVIRONMENT chip_design
```

the version structures are defined by the following commands of level 2

```
DEFINE-GRAPH     FOR chip_design: history_graph, time_list

DEFINE-PARTITION FOR chip_design: consistency
```

After the definition the partition consists of a default class 'null' which comprises all versions not explicitly assigned to a certain class.

The additional classes are introduced by

```
DEFINE-CLASS
FOR consistency OF chip_design : (unchecked, test_a, test_b, cons)
```

The insertion (level 1) of a certain chip object (e.g. for designing a cpu) into this environment is done by

```
DEFINE-OBJECT cpu
IN chip_design
```

When the object is inserted the version structures existing in the environment chip_design are automatically created for this cpu object.

A version is inserted into the database by

```
INSERT-VERS versl
INTO cpu : (x)
```

X denotes the version created by the designer. Its version_id is set to 'versl' by the user. The INSERT-VERS command (level 1) inserts the version into the version set of the object. At the same time it is linked to the structures of the object defined by its environment as follows: it constitutes an independent component in all graphs (i.e. it is not connected with other versions) and is a member of class 'null' in all partitions.

Now let us assume that the newly inserted version 'versl' has to succeed the version designed by author 'Miller' at 1-28-86 in the history graph, i.e. 'versl' was derived from Millers version designed at 1-28-86.

```
CONNECT-VERS
IN history_graph OF cpu
SOURCE :      SELECT-VERS *
              FROM cpu
              WHERE designer = 'Miller' AND time = '1-28-86'
DESTINATION:  SELECT-VERS *
              FROM cpu
              WHERE id = 'versl'
```

SOURCE and DESTINATION determine the versions which are connected. In general SOURCE and DESTINATION are sets of versions, and for every element of SOURCE the set of DESTINATION elements is computed and connected to it.

The insertion at the end of the time_list is accomplished by

```
CONNECT-VERS
IN time_list OF cpu
SOURCE      : SELECT-VERS Y
              FROM cpu Y
              WHERE NOT EXISTS (succ in time_list(Y))
DESTINATION : SELECT-VERS *
              FROM cpu
              WHERE id = 'versl'
```

Finally 'versl' is assigned to the class 'unchecked' by the command

```
SHIFT-VERS
FOR consistency IN cpu
TO unchecked
WHERE id = 'versl'
```

Instead of the above series of commands a more complex INSERT command can be provided at level 3 of the general version concept. This enables the user to insert the version into all structures of the object simultaneously.

The structure of the object cpu after repeated insertions of new versions is as follows:

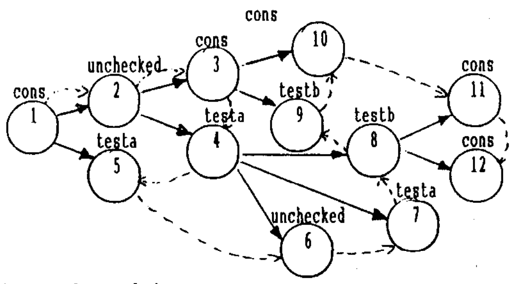


Fig. 1 : The object cpu

The user may not only select versions by attribute qualification but also by exploiting the version structures. Direct predecessors and successors of versions can be found in graphs by means of the operations succ and pred, succ* and pred* are the corresponding transitive closure operations. In partitions the membership of a version in a certain class can be checked.

Thus, the following complex query might be formulated: Find all consistent versions of designer 'Miller' which were created later than 'vers_xy' and have not been developed to newer versions yet.

```
SELECT-VERS Z
FROM cpu Z
WHERE class IN consistency (Z) ='cons'
AND   Z.designer = 'Miller'
AND   (SELECT-VERS *
       FROM cpu
       WHERE id = 'vers_xy') IN (pred* IN time_list(Z))
AND NOT EXISTS (succ IN history_graph(Z))
```

4.2 Advanced mechanisms for adaptation

The above proposed version model supporting graphs and partitions provides possibilities to represent various version concepts needed in practice. But not every individual user needs to see all resulting version structures, and sometimes the general model provides more than the application requires to represent its specific environment. Therefore, level 3 offers operations to adapt the version model to the demands of the environment.

Views

Let us assume, a user wants to work only with consistent versions, and he is interested in the derivation history only. Thus, two version structures of the chip design environment are useless to him. It is convenient to define a view on the basic version set and its version structures which corresponds to his understanding of the object. Fig. 2 shows the view resulting from the following command:

```
DEFINE-VIEW my_cpu
ON chip_design C
WITH elements : SELECT-VERS V
                FROM C
                WHERE class IN consistency (V) = 'cons'
WITH GRAPH derivation-graph
   SOURCE     : SELECT-VERS V_pred
                FROM C
                WHERE V_pred in elements
   DESTINATION: SELECT-VERS V_succ
                FROM C
                WHERE V_succ in elements
                AND V_pred IN pred* in history_graph (V_succ)
                AND NOT EXISTS (SELECT-VERS V_between
                           FROM C
                           WHERE V_between in elements
                           AND   V_between IN
                                 pred* in history_graph (V_succ)
                           AND   V_between IN
                                 succ* in history_graph (V_pred)
```

Within the view object 'my_cpu' versions V_pred and V_succ are connected to each other if V_pred is the predecessor of V_succ in the history-graph and there exists no consistent version between them (V_between).

Note: In this example we use an environment variable 'c' representing any object of the environment.

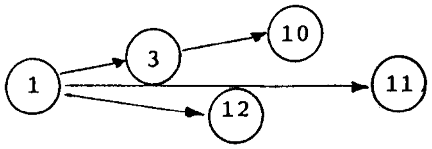


Fig. 2: View with all consistent versions and their derivation history

Constraints

By imposing constraints on operations there are further possibilities to define specialized version concepts. Any command of the general version model may be restricted by a command DISALLOW for all version environments in the system as well as for individual environments only. Operations may either be prevented totally or the restriction may apply to specific states of data only. Thus, DISALLOW is a means to define integrity constraints for version sets and their structures.

If, for instance, no partitions are used in an environment, all operations for partitions will be disallowed. Thus, three simple commands establish a graph oriented version concept.

In our environment it would be convenient to restrict the graph structure time-list to a linear list. The creation of a new edge is prevented if the source version has a successor or the destination version has a predecessor:

```
DISALLOW
  CONNECT-VERS
  FOR time-list IN chip-design X
  SOURCE      : SELECT-VERS Y
                FROM X
                WHERE EXISTS (succ in time-list (Y))
  DESTINATION : SELECT-VERS Z
                FROM X
                WHERE EXISTS (pred in time-list (Z))
```

The operation DISALLOW can also enforce a certain discipline on the handling of partitions. In our example we assume that the design process allows moving of versions between classes only in the sequence unchecked -> test_a -> test_b -> cons, or from any class into unchecked. This is formulated by

```
DISALLOW
  SHIFT-VERSION
  FOR consistency IN chip-design
  TO test_a WHERE class in consistency = test_b    or cons
  TO test_b WHERE class in consistency = unchecked or cons
  TO cons   WHERE class in consistency = unchecked or test_a
```

Any allowable sequence of transitions may be defined in this way. In the following chapter we introduce mechanisms which, in addition, allow to control which program causes a database update. Thus, it can be guaranteed that the transitions are caused only by the respective tools and not by users at their will. This enables the system to ensure operational integrity.

Complex operations

As illustrated by our example it is often necessary to apply a sequence of several basic operations in order to insert a new version into all existing structures. To make this series of operations available as one unit, the single operations can be hidden in a complex operation called transaction.

To provide a high-level insert operation, we can define the transaction CHIP_DESIGN_INSERT, as shown in Fig. 3. The call of

```
CHIP_DESIGN_INSERT (x)
  WITH 'vers1'
  INTO cpu
  AFTER (SELECT-VERS *
        FROM cpu
        WHERE designer = 'Miller' AND time = '1-28-86')
```

corresponds to the sequence of commands in chapter 4. Parameters of transactions are linked by keywords determined by the definer of the transaction (here WITH, INTO, AFTER) to show its semantics.

```
DEFINE-TRANSACTION chip_design_insert (new_version)
                   WITH vers_id
                   INTO obj_id AFTER pred_vers
FOR chip_design

   INSERT-VERS vers_id
   INTO obj_id : (new_version)

   CONNECT-VERS
   IN history_graph OF obj-id
   SOURCE      : pred_vers
   DESTINATION : SELECT-VERS *
                 FROM obj-id
                 WHERE id = vers_id

   CONNECT-VERS
   IN time_list OF obj-id
   SOURCE      : SELECT-VERS Y
                 FROM obj-id Y
                 WHERE NOT EXISTS (succ in time_list(Y))
   DESTINATION : SELECT-VERS *
                 FROM obj-id
                 WHERE id = vers_id

   SHIFT-VERS
   FOR consistency IN obj-id
   TO unchecked
   WHERE id = vers_id
```

Fig. 3: Definition of transaction 'CHIP_DESIGN_INSERT'

Transactions may call basic operations as well as other transactions. Thus, the basic operations can be regarded as the most simple transactions and are also visible at the user interface. In order to provide the user with the very transactions he is allowed to use, access rights are imposed on transactions. In our example no user might, for instance, be allowed to use the basic operation INSERT to ensure that new cpu versions are inserted into all version structures correctly. Versions may be inserted only by the complex command CHIP_DESIGN_INSERT.

5. Embedding application tools

Another aim of the general version model is to guarantee what we call 'operational integrity'. Assume, for instance, that newly designed circuits must be checked by the three simulation programs sim_a, sim_b, sim_cons, in this order. Success or failure of a simulation is expressed by a corresponding transition of the version within the partition 'consistency'.

Operational integrity has not only to guarantee correct transitions but has to ensure that the transitions are caused exclusively by the results of the simulation packages. This requires the system to control the programs causing the transitions between classes, i.e. the programs must be part of the transactions.

In principle, the definition of these transactions demands a tool as powerful as usual programming languages, which provides control structures as branches, loops, subroutines, etc. Several authors offer these mechanisms explicitly with their data models /BrRi/, /KiMc/ /MyBW/ /AlCO/.

For many practical objects, however, the following simple idea may lead to satisfactory results: A version environment is informed about application programs and treats them as transactions. Thereby the operations of the environment are expanded by the applicable design tools. This allows to ensure that the database may only be modified via these tools.

The application programs are stored as string objects in the database. Further information, e.g. documentation, source, etc, may be added. Imported application programs get the status of a transaction, which implies that they may contain other transactions.

To import the simulation programs into our example environment, the following commands are necessary:

```
DEFINE-TRANSACTION simulate_a
FOR chip_design
IMPORT source FROM <file_name1>
       code   FROM <file_name2>
       doc    FROM <file_name3>
```

Corresponding commands import the programs simulate_b and simulate_c.

To summarize, transactions look as follows: Transactions consist of basic operations, subtransactions and imported application programs. The environment administrator places a certain selection of transactions at the users' disposal by means of granting access rights. Access to the data is exclusively allowed by these transactions.

Import of application programs serves two aims: On the one hand, control structures of programming languages may be used in the definition of transactions, on the other, the tools of the application are tied to the version environment. This may be regarded as a step from version environments to 'application environments' which presents themselves as a set of versioned objects with the same version structures and the operations and tools for manipulating them. Thus, an application environment may be seen as a class of an object oriented system defining structure and operations which are inherited by its instances.

6. Representing some version models by version environments

After the description of the general version model, we now briefly show how the specific version concepts of chapter 2 can be implemented by version environments. Obviously the time relationships used in commercial applications, and e.g. in /MüSt/, can be modeled by using version graphs restricted to linear lists.

The version structure defined in /Lock/ may be realized with one version graph. The figures 4a and 4b show two alternative structures of the graph with the second one more explicitly expressing the time sequence of versions. In both graph representations, alternatives and revisions are characterized by respective classes of a partition. Constraints (by DISALLOW) are used to describe the specific appearance of the graph.

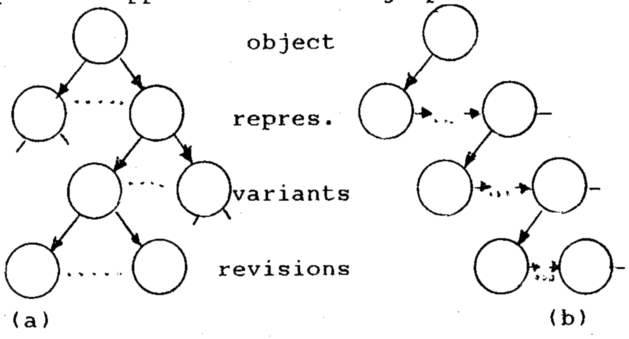


Fig. 4: Graph structure to represent the model in /Lock/

To model the version structures of /KaLe/, a version environment is defined, which has a partition with classes for every version state. Moreover, there exists a version graph in form of a linear list. It orders all versions with respect to their creation time to record the state transitions and to support the time related queries specified in /KaLe/. Operations on the graph and the partition are restricted as to the allowable transitions. For instance, the derivation of alternatives is permitted from in-progress-versions and alternatives only; versions can be moved into class 'released' from class 'effective' only.

The most flexible concept is proposed in /DiLo/. It is the only model known to the authors, which allows the adaptation of the version management to the application. The structure of clusters can be represented in the general version model by one or more partitions. By indirections it is possible to obtain structures similiar to version graphs in /DiLo/, too. But the explicit representation of version graphs in our general version model allows the user to express the ordering relationships of his versions more directly and to deal easier with them. Furthermore the distinction between graphs and partitions seems to be advantageous with respect to the implementation since specific structuring mechanisms can be supported more efficiently at lower levels of the system architecture.

7. Conclusion

In this paper we have introduced a general version model expressed by the concept of version environments. A version environment offers two mechanisms for structuring the version sets of objects: graphs and partitions. By the use of views, constraints and transactions the version environment may be tailored to specific user requirements. The embedding of application tools into the version environment provides the users with their specific application environment which consists of the objects, their version structures and the tools operating on them. The proposed concept is more general and more powerful than the concepts published so far; it has been shown, how well known version concepts can be implemented by means of version environments.

The general version concept is to be implemented in connection with the AIM-project (/Dada/, /Lum/) at IBM Scientific Center, Heidelberg. Their NF2 relations together with a basic time version concept /DaLW/ seem to be a good basis for efficient implementation. The main problems currently are the physical representation of versions and mechanisms to support graphs and partitions at lower levels of the database system by storage structures and access paths.

The version model proposed in this paper cannot stand alone. It has to be integrated into a broader context which considers that objects are very complex, composed of other objects and can be seen from different points of view (representations, see for instance /Neum/). This leads to a three-dimensional concept of (composed) objects, representations, and versions (for objects and representations).

To integrate versions and composed objects, for instance, the basic idea is to have (part-of-)relationships between objects, i.e. sets of versions. To identify the specific versions bound together the structuring mechanisms of the introduced version concept are used. Thus, an object may be composed of specified versions of other 'abstract objects' or is build up by versions identified by denoting the class to which they belong. For instance, an object may be composed of the valid versions of its subobjects. Thus, objects may be composed of subobjects in a very flexible way by using the possibilities of the general version concept.

Literature

/AlCO/ Albano, A., Cardelli, C. and Orsini, R.: *Galileo: A Strongly-Typed, Interactive Conceptual Language.* ACM TODS 10, June 1985

/BaKi/ Batory, D.S. and Kim, W.: *Modelling Concepts for VLSI CAD Objects.* ACM TODS 10, Sept. 1985

/BrRi/ Brodie, M.L. and Ridjanovic, D.: *On the Design and Specification of Database Transactions.* On Conceptual Modelling, Springer 1984

/Dada/ Dadam, P., et.al.: *A dbms prototype to support extended NF2-relations: An integrated view on flat tables and hierachies.* Proc. ACM-SIGMOD, Washington 1986

/DaLW/ Dadam, P., Lum, V. and Werner, H.-D.: *Integration of time versions into relational database systems.* Proc. 10th VLDB, Singapore 1984

/DiLo/ Dittrich, K.R. and Lorie, R.A.: *Version support for engineering data base systems.* IBM Research Report RJ4769 (50628), San Jose 1985

/Härd/ Härder, T.: *Considerations on modelling and integrating time into temporal database systems.* University of Kaiserslautern, Rep. No. 19/84 (in German)

/KACh/ Katz,R.H., Anwaruddin,M. and Chang, E.: *A Version Server for Computer-Aided Design Data.* Report No. UCB/CSD 86/266, University of California, Berkeley 1985

/KaLe/ Katz,R.H.; Lehmann,T.J.: *Database Support for Versions and Alternatives of Large Design Files,* IEEE Transactions on Software Engineering 10 (March 1984)

/KiBa/ Kim, W. and Batory, D.S.: *A Model and Storage Technique for Versions of VLSI CAD Objects.* Proc. "Foundations of Data Organization", Kyoto 1985

/KiMc/ King, R. and McLeod, D.: *A Unified Model and Methodology for Conceptual Database Design.* On Conceptual Modelling, Springer 1984

/KSUW/ Klahold,P.,Schlageter,G., Unland,R. and Wilkes,W.: *A transaction model supporting complex applications in integrated information systems.* Proc. ACM-SIGMOD 1985

/Lock/ Lockemann, P.C., et.al.: *Requirements of technical applications to database systems.* Proc. "Database systems for office, engineering and science", Springer 1985 (in German)

/Lum/ Lum,V.: *Design of an integrated DBMS to support advanced applications.* "Foundations of Data Organization" Kyoto 1985

/MüSt/ Müller, Th. and Steinbauer, D.: *A language interface for version control in CAM databases.* Proc. "GI-Jahrestagung, Sprachen für Datenbanken", Springer 1983 (in German)

/MyBW/ Mylopoulos, J., Berstein, P.A. and Wong, H.K.T.: *A Language Facility for Designing Database-Intensive Applications.* ACM TODS 5, June 1980

/Neum/ Neumann, T.: *On representing the design information in a common database.* Engineering Design Applications, Database Week, SIGMOD 1983

Appendix: Operations of the general version concept

```
DEFINE-ENVIRONMENT <eid>
DELETE-ENVIRONMENT <eid>
```

Level 1:

```
DEFINE-OBJ <oid> IN <eid>

DELETE-OBJ <oid>

INSERT-VERS <vid>
  INTO <oid> : <x>

SELECT-VERS *
  FROM <oid>
  WHERE (qualification)

DELETE-VERS
  FROM <oid>
  WHERE (qualification)

UPDATE-VERS
  FROM <oid>
  SET <x>
  WHERE (qualification)
```

Level 2: Operations for graphs

```
DEFINE-GRAPH
  FOR <eid> : <gid>

DELETE-GRAPH
  FROM <eid> : <gid> .

CONNECT-VERS
  FOR <gid> IN <oid>
      SOURCE:      (version-set of <oid>)
      DESTINATION: (version-set of <oid>)

DISCONNECT-VERS
  FOR <gid> IN <oid>
      SOURCE :     (version-set of <oid>)
      DESTINATION :(version-set of <oid>)

pred  in <gid> (version-set of <oid>)
pred* in <gid> (version-set of <oid>)
succ  in <gid> (version-set of <oid>)
succ* in <gid> (version-set of <oid>)
```

Level 2: Operations for partitions

```
DEFINE-PARTITION
  FOR <eid> : <pid>

DELETE-PART
  FROM <eid> : <pid>

DEFINE-CLASS
  FOR <pid> IN <eid> : <cid,....>

SHIFT-VERS
  FOR <pid> IN <oid>
  TO <cid>
  WHERE (qualification)

DELETE-CLASS
  FROM <pid> IN <eid> : <cid>

Class in <pid> (version-set of oid)
```

Level 3:

```
DEFINE-VIEW <name>
  ON <eid> <obj-var>
  WITH elements : (version-set of <obj-var>)
  WITH Partition <pid> (class:<cid1>,,<cidn>)
       WHERE (cid1 = (version-set of elements)
                :              :
              cidn = (version-set of elements))
  WITH Graph <gid>
       SOURCE      : (version-set of elements)
       DESTINATION : (version-set of elements)
          :

DISALLOW <operation>

DEFINE-TRANSACTION <tid> (keyword)<par>..
  FOR <eid> : sequence of transactions and basic operations
```

Level 4:

```
DEFINE-TRANSACTION <tid>
  FOR <eid>
    IMPORT SOURCE FROM <file_name>
           CODE   FROM <file_name>
           DOC    FROM <file_name>
```