in: Proc. 2nd Int. Symposium on Database Systems for Advanced Applications (DASFAA), Tokyo, Japan, 1991.

A Framework for Version-based Cooperation Control

Wolfgang Käfer

University Kaiserslautern, Department of Computer Science,
P.O. Box 3049, 6750 Kaiserslautern, Germany

Abstract

In engineering environments the support of teamwork is a basic feature which is often not addressed by the design of so-called non-standard database systems (NDBS). First attempts to support cooperation were made by the definition of nested transactions, design transactions, etc. which modify the transaction's inherent ACID-principle. Observance of the ACID-principle guarantees the logical single user mode and isolation which encapsulates the work of the various users: this mode is the contrary of teamwork and cooperation, i.e. the designers of one team work in parallel, use unreliable data and they are aware of both these facts. Thus, we need a system which controls the cooperation by keeping track of data dependencies, rather than a system simulating single user mode.

The use of unreliable data (belonging to one design object) is forced by the long duration of design processes. Accordingly, the objects have to be made accessible at special design states leading to versions of the objects, which provide different features based on their design state. These features are in general a subset of the features which should be provided by the final design object. Introducing these features (and means to detect and control them) into the DBMS gives us a notion of consistency in terms of correctness or completeness of design objects, i.e. a design object is consistent if it contains at least one version which provides all features specified as the design goal. Furthermore, each subset of the features itemizes a distinct level of consistency. Therefore, cooperation takes place by providing and using object versions along with their features, i.e. at different consistency levels. Hence, cooperation control can be focused on controlling object versions and usage relations among them. Thereby, obeying the consistency levels which are provided and which are demanded. Using these consistency levels for conflict detection and resolution leads to meaningful cooperation control.

In this paper we introduce a version model as basis for the support of teamwork. Versions comprise the data belonging to one (complex) design object valid at a particular design state. Furthermore, we introduce the notion of feature sets which we use as a means to detect, achieve, and control the consistency of object versions. We demonstrate which functionality has to be added to the database management system in order to support cooperation.

1. Introduction & Motivation

With the success of database management systems (DBMS) supporting commercial or business environments much effort has been invested into the integration of these systems into a wider range of applications, such as VLSI-design, software engineering, office automation, computer aided design, computer aided manufacturing, etc. This is because they provide

appeared in:
Proceedings of the 2nd International Symposium on Database Systems for Advanced Applications (DASFAA), Tokyo, Japan, April 1991

- multi user mode and
- fault tolerance (recovery mechanisms).

However, an investigation of these various approaches has shown that the data definition and data manipulation capabilities of the conventional DBS are not powerful enough to satisfy the new requirements [Sidl80, Härd83,Härd89], such as long fields, complex objects, recursion, etc. in an appropriate fashion.

Therefore, in the last few years advanced DBS have been developed which cover these modeling issues by some kind of object orientation [Ditt88]. However, the main features of these new systems concentrate on data definition and data manipulation. The new requirements occurring in engineering applications in the fields of multi user mode and recovery mechanisms, such as

- communication and cooperation among designers (teamwork) and
- long duration of design processes,

are often not addressed. For that matter, they are also not adequately satisfied by these enhanced DBS. However, multiple changes of the transaction concept [Beer88,Davi78,Ditt87,Kim84,Klah85,Weik84] have been proposed, because conventional transactions follow the so-called ACID-principle [Härd83], which is in some sense contrary to the above mentioned requirements (atomicity vs. long duration and isolation vs. teamwork). Even the aspect of consistency provided by conventional transactions is more a postulation than it is a guarantee and therefore it is not useful in design environments.

The conflict between the transaction concept and the requirements of engineering design processes is essentially based on the totally different environments they belong to. In business applications, transactions serve as a means to simulate "single user mode" in an *automatic* manner, i.e. transactions are either completely executed or they have no effect on the database. Furthermore, a schedule of transactions will be correctly synchronized if, and only if a sequence of these transactions producing the same result exists. These basic fundamentals of the transaction concept are not present in engineering design environments. An engineering design process [Banc85,Kelt87,Kort87,Lorie83, Rant89,Rehm88,Lian90] is not atomic. Firstly, in the event of a failure it is not appropriate to back out many day's work in order to get ("all" or) "nothing". Secondly, a sequential schedule of design processes is in opposition to teamwork, i.e. to cooperation. A designer *cooperates* with other designers by accepting hints from them, how his own design should proceed and by giving hints to them as to what features of their design are important or useful for his own work. Therefore, the exchange of uncommitted (i.e. incomplete or inconsistent) information about the design is a definite requirement for cooperation. In order to control the cooperation process [Fisc89,Ditt87, Vanh90,Nodi90,Skar90], we have to provide a means for the description of "consistency levels", i.e. of "inconsistent or incomplete" design objects. We do this by the notion of "features" of design objects and design object versions. For the moment you can think of "features" like the specification of interfaces for example in the area of software engineering. The features specified by design object describe the goal of the design, i.e. all modules along with their interfaces. Each version of the design object provides only the subset of features (i.e. only those interfaces) which are actually implemented. Furthermore, other detailed information about the actual implementation which might be useful for the users of the interfaces, i.e. a more detailed description of the value range of parameters supported by the actual implementation or some information about applied tests to the module. Thus, each version of the design object represents a certain design stage itemized by the features of the version.

Furthermore, we have to redefine the notion of "conflict" known from traditional transaction models. E.g. a *conflict w*.r.t. cooperation arises, when features of a design object are altered (i.e when interfaces are changed), which were actually required by other designers. As a consequence, the conflict resolution strategies chosen in traditional transaction models in order to guarantee a sequential schedule are not appropriate for design processes: in the event of a conflict, the designers have to come to some agreement as to how the whole design should proceed (i.e. how the interfaces should really look like). Thus, the conflict resolution must be made by the designers themselves. Our proposed database system accepts the result of the conflict resolution, performs the appropriate actions, and keeps track of all these actions.

The remainder of the paper is organized as follows. Section 2 provides a little bit of the flavor of how design is done. We first introduce some principles of design organization and sketch the basic schedule of a design process. Then we give a short overview of a related processing model in a server/workstation environment. Section 3 introduces the framework for cooperation control. We start with the description of the version model which allows the definition of complex object versions as well as the definition of features of these versions. Since these features describe the design state of the versions, we can use them to control cooperation. This is shown in section 3.2. We close with a summary and an outlook as to our future work.

2. Basic principles of design

Complex engineering objects, e.g. a complex program in the area of software engineering or a chip in the area of VLSI design are designed by a group of people, who work cooperatively rather than competitively. Each member of the group work on a single part of the whole design object. The design process lasts for hours, days, or weeks. During the design process the designer produces *states* of the design object as special steps of the design stage. These design states are represented in the DBS as *versions* of the object. While working on an object, e.g. a module or the cpu, the designer wants to control whether another designer may or may not see or use versions of it. On the other hand, he wants to *see* (get some information about the work of other designers) and perhaps to *use* versions of objects (i.e. integrate them in his own work), which are currently under design by his colleagues. Furthermore, he wants to get some idea as to what parts of the design are relatively stable and what parts are (at this point in time) still in the drafting phase, i.e. he needs to know, what features (what interfaces are realized, and what interfaces are only realized by "dummies"?, what tests have been applied?, what input is for the actual realization appropriate, and what input will produce failures? etc.) are currently provided by the design object (i.e. of particular versions of the design object) at the current design state. However, each designer knows that he is working with "unreliable" data, which are continuously being changed by colleagues. Therefore, in order to organize his own work he needs to know

- how reliable the data are,
- what features the data offers (at a certain design state),
- who is designing that piece of work.

Furthermore, he must have the assurance, that the data are stable during the period over which he uses them. In the following two subsections, we will detail this view of design with regard to the organizational and executional aspects.

2.1 A logical view to design processes

In order to achieve the goals mentioned above, we have to take a closer look at how design is organized. The design, e.g. of a chip, is a very complex task. So, this *design task* is (recursively) divided into a number of simpler design tasks (subtasks) forming a *design task hierarchy*†. In our example (cf. Fig. 2.1), the design of a chip might be divided into a number of subtasks, e.g. the design of a cpu and the design of a memory. Both subtasks share a common subtask for the design of a bus. Furthermore, a floating-point unit is integrated into the cpu. The division of the design tasks in subtasks implies two major aspects:

- **design object decomposition**
  As mentioned above, the result of each design task is a so-called *design object* (DO), e.g. representing a chip in the area of VLSI design. The design object associated with a specific design task is the composition of the DO of this task itself and the DO of each of its subtasks. In our example, a chip is the composition of a cpu, a memory, a bus and a floating point unit. Therefore, the design task hierarchy describes the schema or in some sense the methodology of how to design a chip. The (composite) DO are instances of this schema. Furthermore, the design task hierarchy serves us as a structured and application-oriented view of the teams of designers assigned to the design tasks. Generally speaking, each designer has to cooperate with the designers assigned to the direct predecessors and the direct successors of his own design task. In our example the designer of the memory has to cooperate and therefore to communicate with the designers of the chip itself and the designer of the bus, but not with the other designers involved in the whole design task.
- **cooperative work**
  All subtasks within a design task hierarchy can be performed in parallel. However, the resulting DO must fit together, i.e. the subtasks of one specific design task have to exchange information about their DO in order to build up the composite DO. Due to the long duration of the design process the designers have to make intermediate results of their design (i.e. versions of their DO) accessible for their colleagues. However, these versions have to be meaningful for the colleagues, i.e. the versions have to provide the consistency level they belong to. Fig. 2.2 depicts an example of a design process along with the produced object versions (and the related features describing their design state or respectively their consistency level).

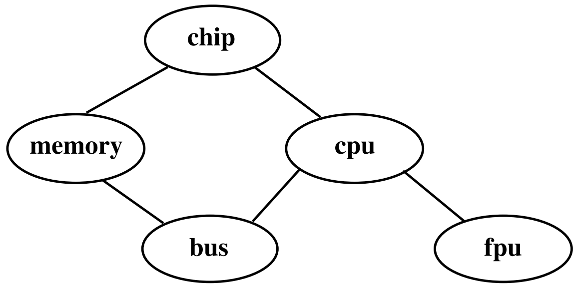


Figure 2.1: Example of a design task hierarchy

† We use the term "hierarchy" in a more generalized fashion, i.e. we allow for multiple predecessors of a subtask.

So far, we have described the passive issues of the design, i.e. the structure of the design tasks and the composite design objects, which give us a continuum for cooperation. But, we cannot control cooperation without the active elements of the design, i.e. the designers themselves. The activity of the designers affects the DBS in two ways:

- **design process**

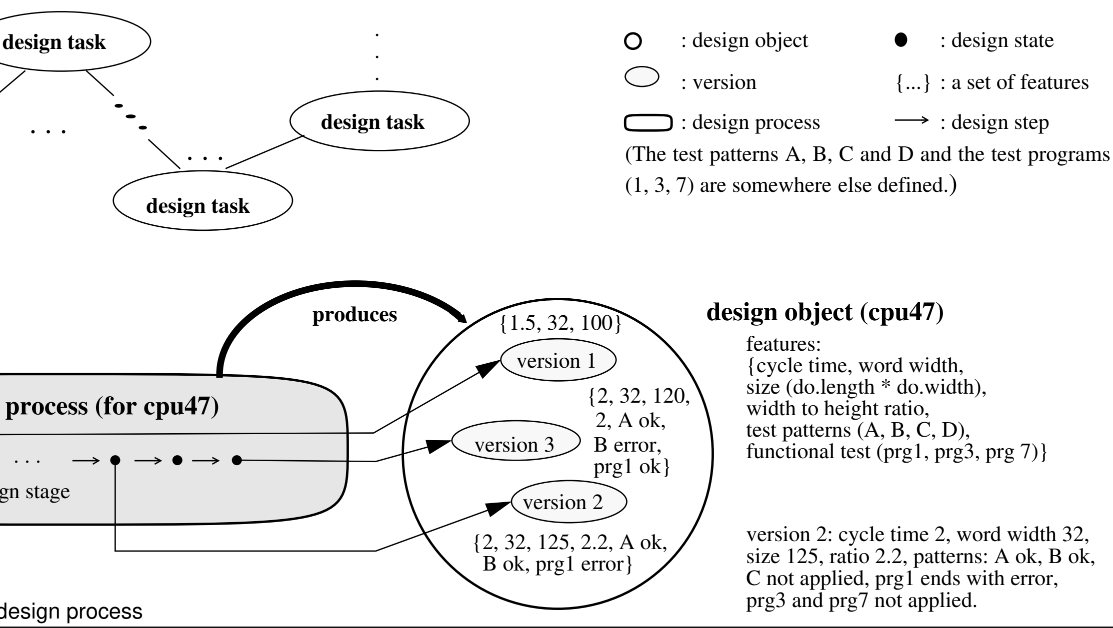
  When a designer is assigned to a design task, he starts a *design process* (DP) (cf. Fig. 2.2) in order to generate a specific DO, say cpu47. During the design process the designer gets a couple of DO from the database and produces himself a couple of versions belonging to one DO (e.g. cpu47). DP are the run units of our system similar to transactions in conventional DBS. But opposed to transactions, design processes do not follow the ACID principle. Furthermore, DP are started from designers only, i.e. by real people and not by programs etc. This is necessary, because of our conflict resolution strategy.

- **conflict resolution**

  As mentioned above, designers work in parallel, use unreliable data and are aware of both these facts. If a designer uses unreliable data (i.e. a version of a DO not yet released) owned by a colleague and the version is to be changed, the DBS informs both designers of the conflict. The conflict resolution itself has to be made by the designers themselves, i.e. the change has to be revised (e.g. a new version from the object is built) or the designer uses the new version. During the conflict resolution phase, the DBS provides both the old and the new version. Furthermore, the DBS has to keep track of the conflict resolution process, especially of its result.

Summarizing, the engineering design is characterized by the following terms:

- Design tasks are divided into many subtasks.
- Each design task is performed by one designer (leading to a design process).
- Designers assigned to subtasks (within the design task hierarchy) work cooperatively together.
- The result of a design process is a design object.



- Each design object may exist in many versions.
- Each version is specified by a set of features.
- The designers know, that
  - they work in parallel and
  - use unreliable data.

Up to this point, we have introduced the basic concepts of design and cooperation grasping parallel design processes and the usage of object versions as the basis for cooperation. However, this logical model of the design cannot be viewed independently from the underlying software and hardware architecture. Since we do not want to detail or to discuss specific architectures, we outline only a typical design environment.

2.2 A physical view to design environments

In this chapter, we characterize the software and the hardware architecture in which design takes place [Dada88,Depp87,Härd87b,Härd88a]. We assume a workstation-server architecture as shown in Fig. 2.3. The NDBS running at

and descriptive query language which al-
ne manipulation of complex objects
2,Paul87,Dada86,Care86,Lori83,Lori84].
ace can be tailored to the needs of specific
gn, geometric modeling etc. The version
complex-object versions, i.e. of creating
the derivation graph, etc. Moreover, the
ns for handling the features of complex-
used by the design manager to control co-
ne correctness or the completeness of the

ata preparation layer managing the data
server thereby transforming the object
epresentation of the data (and vice versa).
in a special main memory structure guar-
omparable to pointer-references in pro-

he processing spans three phases:

...t of object versions (of different objects) from the server thereby demanding sets of features of these object versions. The sets of features serve as a notion to describe the minimal correctness or completeness of versions causing them to be useful for the designer. After selecting the appropriate versions and testing the related authorization and access conditions as well as determining whether the data are already checked out in an incompatible mode, we start a conventional transaction in order to perform the check-out operation. Therefore, check-out operations (and equally check-in operations) are atomic actions and strictly isolated from each other. Access, authorization, and cooperation control are completely managed within the server.

- **processing**

  During the processing phase which is characterized by very high access frequencies to the design data, this data has to be maintained in an object buffer, i.e. in main memory. We can structure the processing using the following commands [Härd88b]:

  - SAVE: the current design stage is captured in non-volatile storage

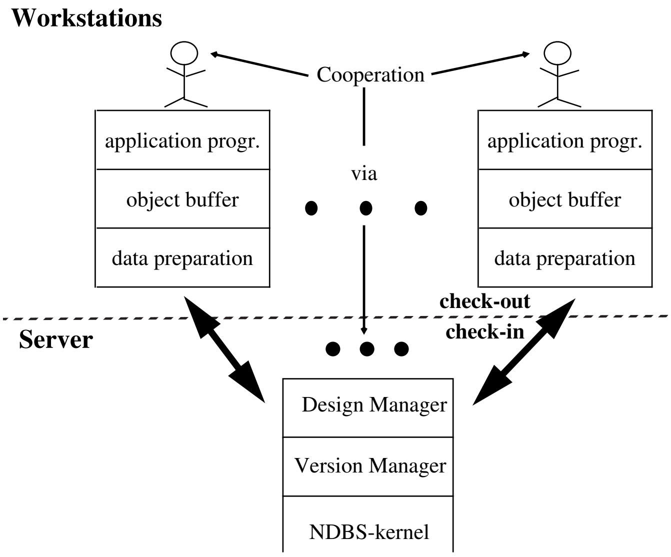


Figure 2.3: A workstation-server architecture for engineering environments

- RESTORE: backing out to a previously saved design stage
- SUSPEND/RESUME: interrupting and continuing the processing (e.g. take a coffee break)

During the processing, new versions and/or new objects can be generated. This data is held only into the workstation and thereby only visible (and accessible) to the designer using this workstation. The only way to make these data visible and accessible in a particular form to other designers is via the server, i.e. a check-in operation has to be performed.

- **check-in**

  As mentioned above, the check-in operation is performed as a conventional transaction. During the check-in processing all authorization, access, and consistency rules concerning the data have to be checked. If none of these conditions is violated, the check-in operation can be performed, otherwise the check-in is revised leaving the data at the server site unchanged. The designer is informed about the consistency violations. Since the data concerning the check-in operation is not lost (i.e. it is held at the workstation site), the designer can correct the data or alter other parameters of the check-in operation in order to integrate the refined data into the DBS.

Cooperation is characterized by two aspects with respect to this design environment. Firstly, the only way to cooperate is via the central server. This implies that authorization, access, and consistency checks are performed on the whole database. Secondly, cooperation is controlled by the design manager using the set of features which is assigned to the version at the check-in phase and the (probably different) set of features which is claimed at the check-out phase.

3. The Framework

In this chapter, we will introduce a framework which allows the description of design objects, versions, features, and usage in a more formal fashion. Since, we do not want to introduce a specific data model, we do not consider the internal structure etc. of design objects. Firstly, we introduce the notion of features, which allows the definition of private, tentative, reliable and released versions as well as alternatives and variants of design objects (section 3.1). Afterwards, we demonstrate how we can use these notions to describe cooperation (section 3.2).

3.1 The Version Model

Design Objects

Design objects are characterized by three different aspects. Firstly, a design object is a complex object, i.e. it consists of a number of more elementary objects and a number of links among these objects. The specific representation of a design object is determined by the data model which we will not further discuss in this paper. Secondly, design objects are connected via aggregation in order to build more complex design objects (cf. design task hierarchy). Thirdly, design objects have to meet some kind of **design specification**, i.e. they have to fulfil a number of requirements, such as

- design goal, i.e. what is the purpose of the design object?
- design restrictions, i.e. the maximum size, weight or price of the design object,
- correctness, i.e. what proofs or tests are appropriate for the design object?
- organizational status, i.e. "ready for production" or other states defined by the enterprise.

We refer to those requirements which are known by the system as **features**. Some of these features can be seen as consistency rules, others may be described as algorithms, and still others may be stated by an authorized person. We do not further detail the description of the features, since this subject is beyond the scope of this paper. However, we can now define a **design object** (DO) as

- a complex object with
- a set of features (**{f}**).

In general, the set of features describes the design specification of the design object. We will refer to this set of features as **{f}ds**. **ds** denotes a projection-operator which reduces the related feature set to features belonging to the design specification of the object, i.e. belonging to MapDO(DO) = DO{f}. This set of features comprises **basic features** of the design object as well as **optional features** (in short terms **{f}ds = {f}dsb** ∪ **{f}dso**). The basic features {f}dsb have to be fulfilled by the design objects (e.g. the instruction set of the cpu, the size of the memory, the overall geometrical size of the chip, etc.), whereas the optional features {f}dso describe only goals which have to be realized as well as possible (e.g. the width to height ratio of the chip should be 3:1, the price should not exceed $20, etc.). Furthermore, we can now view a design task as an order to generate a design object specified by a certain set of features or simply, to generate **DO{f}ds**. Thus, we can refer to {f}ds as the specification of the design task.

definitions:

- A **feature** characterizes a special aspect of the design object or of a design object version which is important for the design task. For our framework we can restrict features to names which we will denote as f_i, $i \in \{1,...,n\}$.
- We call the complete set of all features the **definition set**, denoted as **Def_f** = $\{f_i\}$, i = 1 ... n.

functions:

We distinguish between two functions, which map a (sub-) set of the definition set to specific design objects or alternatively to design object version.

function: MapDO: DO → {Def_f}; DO: design object
assigns to a design object an element of the power set of the definition set. Often we abbreviate the result of MapDO (DO_i) as DO_i\{f\}.

function: MapV: V → {Def_f}; V: design object version
Analogously, we abbreviate MapV(V_i) as V_i\{f\}.

projections:

We will introduce a lot of projection operators. In general, a projection operator selects a subset of a given feature set.

projection: x: Def_f → Def_f; reduces a set of features FS ∈ {Def_f} to another set of features fs with fs ∈ {Def_f}.

Often we abbreviate the result of x(FS) as FSx. The combination of the function- and projection operators leads us to terms like x(MapV(V_i)) or abbreviated as V_i\{f\}x. Furthermore, the projection operators belong to specific design objects, i.e. we have to mark the projection operators with their related design objects, denoted as X_{DO}. In order to simplify our notation we don't mark the projection operator if the connection to the design object is unambiguous.

In order to execute a design task, a design process is initiated. The design process will last hours, days, or weeks according to the complexity of the design task. During the design process the design object passes a number of *(preliminary) design states*. The design objects at these stages will obey only a subset of the features related to the design task. We call these preliminary design objects **design versions** of the design object or simply, **versions**. Clearly, versions always belong to exactly one design object and have the same structure as their design object.

Versions

Initially, versions must not obey any of the features related to their design object. However, they must be of the same complex object type as their design object and therefore, they have to satisfy the rules concerning the complex object definition (i.e. they have to be "consistent" in the sense of the complex object data model). Since design processes are goal-oriented, the versions (V) of a design object generated during the design processes will in general have growing sets of features, denoted as **V{f}**. According to the set of features provided by a version we can differentiate among private, tentative, reliable, and released versions

→	**V{f} = { }**	:	**private** version
→	**V{f} ⊂ DO{f}ds**	:	**tentative** version
→	**V{f} ⊇ DO{f}d**s	:	**reliable** and **committed** versions

Private versions do not provide any features, so they have no meaning for other designers. Therefore, they are not visible or accessible for other designers. **Tentative versions** offer a subset of the features specified by the design object. Thus, they are valuable for those designers who rely essentially on these specific features. However, tentative versions are subject for changes in order to meet the full design specification. **Reliable versions** fulfil the basic features {f}dsb and provide the optional features {f}dso of the design specification, i.e. the design of these versions has reached its goal thereby finishing the design phase. However, reliable versions may be changed in order to improve some of the optional features {f}dso or because of other reasons. In many applications there is a need to prohibit some versions from any changes (e.g. versions of design objects which are ready for manufacturing). We call reliable versions which can no longer be changed, **committed** or **released versions**. Often, versions have to pass a specific *release action* (some kind of protocol defined by the enterprise) in order to become a released version.

As you noticed, we use "⊇" instead of "=" in the definition of reliable and released versions. This is due to **additional features: {f}a ⊄ {f}ds.** During the design process, the designer achieves the design specification by making many design decisions which lead to an increasingly more concrete and detailed design object. Some of these design decisions have impact on the overall design object. Therefore, they may be important for other designers. We call the features related to these design decisions additional features or simply, **{f}a** (e.g. the chip will be operable in a high range of temperatures, the chip will be robust against oscillations of the power supply, etc.). Thus, a version (V) of a design object provides two sets of features:

V{f} = V{f}ds ∪ V{f}a; V{f}ds ⊆ DO{f}ds ∧ V{f}a ∩ DO{f}ds = { }

Using these terms, we can easily define alternatives and variants.

projection operators:

ds: features of the design specifications
dsb: basic features of the design specification
dso: optional features of the design specification
a: additional features of the design specification

The following integrity constraints hold for a set of features {f}:

1) {f} ds = {f} dsb ∪ {f} dso
2) {f} dsb ∩ {f} dso = { }
3) {f} ds ∩ {f} a = { }

Alternatives and Variants

Generally speaking, alternative solutions are solutions which are equivalent in the sense that they solve the problem and which are different with regard to the methods or other things they use. In our terminology, we can define "alternative" more precisely: design versions of one design object are **alternative with respect to** a set of features {f}, if they provide at least these features: {f} ⊂ V{f}. The special case, that {f} describes the specification of the design object, i.e. {f} = {f}ds, we call these alternative versions

→ design alternatives: **V{f}ds = DO{f}ds**

Design alternatives, or simply **alternatives**, are special versions which are alternative with respect to the design specification. As mentioned above, versions provide an additional set of features {f}a which describes specific characteristics of the related version. We call the versions which obey the design-specification and provide special characteristics

→ design variants: **V{f}ds = DO{f}ds ∧ V{f}a ≠ { }**

or in other words: *all design alternatives which provide additional features* are called **design variants**.

Based on the notion of design objects, design versions (including alternatives and variants), and sets of features, we can define a set of operations to retrieve and to manipulate each of them.

Operations on design objects:

- **Create / Delete design object**

  Create generates an empty design object, i.e. a design object without any version. Furthermore, this operation defines the object's structure (aggregation relation, elementary objects and links among them), the structure of the derivation graph (e.g. list, tree, directed acyclic graph), and the set of features, i.e. the object specification. **Delete** discards a design object including all versions of the design object.

- **Select <design objects>**

  selects design objects on behalf of their attribute values, their versions, of their features.

Operations on design versions

- **Create/Delete <design versions>**

  generates/discards a design version. The creation of a version leads always to a private version. Committed or released versions cannot be deleted.

- **Select <design versions>**

  selects design versions within one design object on behalf of their attributes or their features.

- **Connect/Disconnect <design version> To/From <derivation graph>**

  These operations maintain the derivation graph of a design object; the structure of the graph and other constraints (e.g. the fan-out of tree-node is less than 5) are sustained.

- **Upgrade <design version>**

  The data and the set of features of the version are changed.
  In general, we assume that the version is replaced by a newer and therefore, by a more complete one. Thus, we **call an upgrade**:

  - **consistent upgrade**
    (if $V\{f\}ds\text{-}old \subset V\{f\}ds\text{-}new$)
    The new versions provide more features of the object-specification than the old one; the additional features $\{f\}a$ may differ.

  - **safe upgrade**
    (if $V\{f\}ds\text{-}old \subset V\{f\}ds\text{-}new \wedge V\{f\}a\text{-}old \subset V\{f\}a\text{-}new$)
    The new versions provides more features (including additional features) than the old one.

  - **corrective upgrade**
    (if $V\{f\}\text{-}old = V\{f\}\text{-}new$)
    The data of the version are replaced by the new one. However, the set of features of the version remains unchanged, i.e. all features remain valid with the new data.

- **Downgrade <design version>**

  The data and the set of features of the version are changed.
  Contrary to the upgrade operation, the new version provides less features than the old one. Analogously, we **call a downgrade**:

  - **non-consistent downgrade**
    (if $V\{f\}ds\text{-}old \not\subset V\{f\}ds\text{-}new$)
    Some features of the design specification which were provided by the old version are no longer provided by the new version.

  - **uncertain downgrade**
    (if $V\{f\}ds\text{-}old = V\{f\}ds\text{-}new \wedge V\{f\}a\text{-}old \not\subset V\{f\}a\text{-}new$)
    Some additional features are no longer provided by the new version; the features related to the design specification remain unchanged.

Operations on features

- **Define <feature> Control is <. . .>**

  defines a feature and gives the DBMS a means (i.e. rules, a procedure, an "authorized person", etc.) to control the feature.

- **Assign/Remove <feature> To/From <feature-set>**

  assigns/removes the feature to/from a set of features of a design object or a design version.

- **Discard <feature>**

  removes the feature from the DBMS, if it is not assigned to a feature-set.

Up to this point, we have introduced a version model which allows the definition of (complex) design objects along with application-oriented versions, alternatives, and variants based on application-oriented features. Furthermore, we have introduced a couple of specific operations on versions such as upgrade, downgrade, correction, etc. In the following chapter, we will show, how we can use this version model to describe cooperation.

3.2 The Cooperation Model

Remember, we have introduced cooperation as the mutually affected design of two (or more) designer, i.e.

- one designer (**the creator**) produces versions of his design object which
- another designer **(the user)** needs in order to complete his own design task.

The creator

The creator of a version has the task to produce at least one version of this design object which fits the design specification. Often this design object is one part of a more complex design object (e.g. actually designed by the "father" design task, cf. Fig. 2.1). Thus, the creator is asked by other designers to make intermediate results of his own design visible to them, i.e. to provide versions of his design object along with their features. Doing this, the creator waives some of his rights on this version, i.e. changing or deleting this version has to be adjusted to comply to the wishes of the users of the version. However, the creator is the only person who has the right to do this. Besides the provision of versions the creator receives hints from the users as to what features of his design are actually needed by the users (i.e. he has to look at the usage specifications related to his versions). Since the creator wants to cooperate with his colleagues in order to reach a common design goal, he will try to provide the required features as soon as possible. Furthermore, the creator is responsible for the complete version set of his design object along with the related feature sets. In particular, finishing his design requires that only committed versions (and therefore consistent versions) are left. Thus, the creator is interested in keeping his version set to a minimum, i.e. he is interested in using the upgrade operation instead of the creation of new versions. Often the creator has to integrate other design objects in his own design object and therefore becomes a user.

The user

The user has to integrate a design object (i.e. a version of the design object) into his own design. Related to the design state of his own design the user needs only a version of the used object which provides a subset of the features describing the design specification. If he uses such a tentative version, he wants the assurance that this version won't be changed or deleted without his agreement, because his own design relies fundamentally on this version. Furthermore, he wants to give hints to the creator of the version what features he will need to proceed with his design. However, using a version does not allow for the change of the version by himself.

As mentioned previously, the design processes of the creator and the user have to be active in order to cooperate, because otherwise a mutual affection is not possible. Summarizing, we can characterize cooperation as follows: *The creator influences the design of the user* by generating versions with growing sets of features: the users of the version will concentrate on such parts of their own design which rely mainly on features of the used version which have been already realized. However, *the user will influence the design of the creator* of the used version by demanding specific features which should be realized next. Thus, the interchange of preliminary object versions and of hints how the whole design should proceed is the gist of cooperation.

The cooperation process

Users have *to integrate versions into their own design object's versions*, which provide a specified set of features. Therefore, they have to look for versions which are suitable for their own design, i.e. for versions which provide at least this set of features. Often the continuum for the search is limited to a "catalogue of standard objects" and objects which belong to their project hierarchy, i.e. cooperation is only possible among siblings and/or father and children of the project hierarchy. The cooperation is initiated by establishing a usage relation between two versions thereby specifying the set of the demanded features. We call this set of features **usage specification** or simply, **{f}us**.

projection operators:	
us:	features of the usage specifications
usb:	basic features of the usage specification
uso:	optional features of the usage specification

The following integrity constraints hold for a set of features {f}:

1) $\{f\}\ us = \{f\}\ usb \cup \{f\}\ uso$
2) $\{f\}\ usb \cap \{f\}\ uso = \{\ \}$

Analogously to the design specification, we can split the usage specification in **basic features {f}usb** which have to be satisfied by the used object version and **optional features {f}uso** which should be realized next or which have to be fulfilled as well as possible (in short terms $\mathbf{\{f\}us = \{f\}usb \cup \{f\}uso}$). Looking at the features demanded {f}us and the features provided by a version V{f}, the DBMS can recognize a lot of situations and perform the appropriate actions.

- $\mathbf{\{f\}usb \subseteq V\{f\}dsb \wedge \{f\}uso \subseteq V\{f\}dso}$
  This version fits; it should be released at least with V{f}us, in order to forbid any further changes of that version.
  → The DBMS should inform the creator of the version, that the version should be released.
- $\mathbf{\{f\}us \subseteq V\{f\} \wedge \{f\}usb \cap V\{f\}dso \neq \{\ \} \vee \{f\}usb \cap V\{f\}a \neq \{\ \}}$
  the user of the version relies on optional or additional features of the version, thereby appointing these features.
  → The DBMS should inform the creator of the version that some of the optional or additional features ($\{f\} = \{f\}usb \cap V\{f\}dso) \cup \{f\}usb \cap V\{f\}a)$) are urgently needed by the user of the version. Thus, these features should be integrated into the design specification.
- $\mathbf{\{f\}usb \not\subset V\{f\} \wedge \{f\}usb \subseteq DO\{f\}dsb}$
  the user of the version needs some features which are a part of the design specification, but which are not provided from the version at this time, i.e. the user gives some hints to the creator of the version which features should be realized next.
  → The DBMS should inform the creator of the version that the user needs a set of features $\{f\} = \{f\}usb \cap DO\{f\}dsb$.
- $\mathbf{\{f\}uso \cap DO\{f\}ds \neq \{\ \}}$
  The user requests some features ($\{f\} = \{f\}uso \cap DO\{f\}ds$) which are not part of the design specification.
  → The DBMS should inform the user that he has to file a change of the design specification.

So far, all these actions are invoked by the user of versions, i.e. by connecting two versions via a usage relation. On the other hand, we can think of actions which are caused by the creators of versions, i.e. the production of a version with an extended set of features or the up-/downgrade of used versions is an interesting event for users. Thus, the users are interested in getting information about the production of new versions which are meaningful for them (i.e. more of the features of {f}us are satisfied by the new version). However, the creators are concerned to inform the users about new versions as well. Firstly, this results from the principle that a design can only be finished if all produced versions are committed. Therefore, all private and tentative versions have to be deleted at the end of the design. Secondly, the creator of the versions has the own interest to keep his version set minimal in order to cope with the interdependencies among his versions, the various sets of features related to the versions and the users of the versions. Of course, deletion, upgrade, or downgrade of (used) versions have to be adjusted to the wishes of the users of the version. Again, the DBMS can recognize a lot of different situations:

- **safe upgrade**
  Since the used version is replaced by a more complete one, it suffices to notify the users about the upgrade. Furthermore, the DBMS can indicate the new features. This can be detailed to really useful features, i.e. to features which were demanded by the user.
- **consistent upgrade**
  Since the new version is closer to the design specification than the old one, the version will be replaced and the users will be notified about the change. However, optional or additional features might be changed. Thus, the old version is kept by the DBMS until all users agree to the change. This gives the users the possibility to adjust their design to the new version.
- **downgrade**
  Since some features of the design specification are lost in the new version, the downgrade operation requires the agreement of all users of the version. The operation is postponed until all users have agreed.
- **deletion**
  Private versions can be deleted at any time, whereas committed versions are not deletable at all. Tentative versions can be deleted if they are not used, otherwise the operation is postponed until all users have agreed.

We do not detail further possibilities due to space limitations and proceed to aspects concerning the control of cooperation. As mentioned above, the DBMS often informs the designer about changes of the design environment or postpones operations which would cause a conflict among the designers. In the following, we will concentrate on the conflict resolution process and the related communication process among the designers and show, how the DBMS can support these processes.

The conflict resolution process

So far, we have outlined a version model which supports multiple levels of consistency by the notion of feature sets as well as a number of operations maintaining versions and feature sets. The exploitation of this knowledge along with the distinction between creators and users of versions allows the DBMS to detect different kinds of conflicts. However, the DBMS is not capable of solving the conflicts by itself. Firstly, backing out the conflicting operations would destroy the hours or days of a designer's work. Moreover, deleting a part of a complex object (on which the conflict had occurred) often violates the consistency of the whole object in an unpredictable way. Secondly, the DBMS has not enough knowledge to solve the conflict by going forward, i.e. by adjusting the objects. Thus, the DBMS can only support the conflict resolution process, i.e.:

- **conflict detection**

  The DBMS can detect different kinds of conflicts based on the usage relation, the feature sets ({f}ds and {f}us) and the operations.

- **conflict resolution**

  The DBMS freezes the environment of the conflict. Thus, revoking of the operation which caused the conflict or adjusting the data (with knowledge of the conflicting operation) is possible. The *notification service* of the DBMS initiates a communication process among the participants of the conflict. The conflict resolution has to be done by discussion among these participants or by direction of a responsible manager. The outcome is passed to the *book-keeping service* of the DBMS. This can be performed by sending a message to the DBMS (e.g. on agreement to an upgrade operation) or by manipulating the data (e.g. removing a usage relation).

- **notification service**

  The DBMS notifies all participants of a conflict, i.e. each participant is informed about the other participants, the versions on which the conflict has occurred and the operation which caused the conflict.

- **book-keeping service**

  Firstly, the book-keeping service takes notes about conflicts (participants, versions, operations) and their outcome. This includes checking the outcome for plausibility, i.e. checking the adjusted feature sets ({f}us ↔ {f}ds) or checking wether a usage relation has been removed. Secondly, the book-keeping service is responsible for the actions which have to be done after conflict resolution, i.e. performing postponed operations, etc.

Thus, the DBMS (i.e. the design manager, cf. 2.3) has to be able to use a notification service like electronic mail to send messages to designers and to receive messages from designers. Furthermore, the DBMS has to serve as an information center about open conflicts, duration of conflict resolution processes, the actual state of a conflict resolution process, etc.

4. Summary

The employment of DBMS in design environments was the motivation to improve the data modeling capabilities of these systems. However, the data processing facilities of the DBMS were not adjusted. Thus, the transaction concept which was developed to guarantee logical single user mode didn't fit the requirements of cooperative design environments. In particular, the notion of consistency within the transaction concept is opposed to the situation in design databases, which are in an inconsistent state over a long time. Thus, we have to define a new notion of consistency in order to allow the DBMS to support cooperative design processes.

It is a very difficult task to define the consistency of a design object in a formal way and, moreover, to define different consistency levels for (the versions of) the design objects. Thus, we have introduced the notion of **feature sets**. Firstly, we use feature sets to describe the goal of the design, i.e. we use them as a means for the design specification. We can easily differentiate among **basic**, **optional** and **additional** features. Any subset of the design specification forms a distinct consistency level, i.e. the consistency of a design object version is described by a subset of the design specification. Thus, design of an object can be depicted as the *task to generate versions of the object with growing (sub)-sets of features* until the design specification is reached. The depiction of the design task even holds for cooperative design, by which the design task is spread over several designers. Two or more design processes cooperate, if one process produces *an object version (along with a set of features) which is used (i.e. integrated) by other processes.* Thereby the users demand specific features which have to be provided by the used version (again differentiated in basic and optional features).

Conflicts occur among (cooperative) design processes as well as among transactions if the basic consistency rule is violated, i.e. if the serializibility of transactions (often indicated by a read/write conflict) or *the growth of a version's feature set is affected*. Analogously to the read/write operations in transaction systems all operations on feature sets are subject for conflicts in cooperative design environments. We have introduced the **upgrade** (**safe**, **consistent**, **corrective**) and the **downgrade** (**non-consistent**, **uncertain**) operation as the means for cooperative design. Contrary to transaction systems there is no way to solve conflicts in an automatic manner (i.e. backing out is not applicable and going forward (making the data consistent) is usually not possible). Thus, the *DBMS can only assist the designers by cooperation*. The DBMS detects conflicts, saves the environment of the conflict and initiates a communication process among the participants of the conflict. The conflict resolution itself has to be done by the designers themselves. Thus, the DBMS serves as notification and book-keeping center.

Our future work will concentrate on acquiring experience in this framework. Therefore, we will have to take a closer look to the description of features and means to control them. In particular, the representation of these means within the DBMS and their efficient execution poses a lot of problems. Furthermore, the point in time when these means should be executed have to be carefully selected.

5. Literature

Banc85 Bancilhon, F., Kim, W., Korth, H.F: A Model of CAD Transactions, Proc. of the 11th Int. Conf. on VLDB, Stockholm, 1985, pp. 25-33.

Bato85 Batory, D.S., Kim, W: Modeling Concepts for VLSI CAD Objects, ACM TODS, Vol. 10, No. 3, sept. 1985, pp. 322-346.

Beer88 Beeri, C., Schek, H.-J., Weikum, G: Multilevel Transaction Management, Theoretical Art or Practical Need?, (Ed.: Schmidt, J.W., Ceri, S., Missikoff, M.), Int. Conf. on Extending Database Technology, Venice, Italy, Advances in Database Technology - EDBT '88, Springer-Verlag, Lecture Notes on Computer Science, 303, mar. 1988, pp. 134-154.

Blan87 Blanken, H., Ijbema, A: Storage of Versioned Objects in a CIM Environment, Proc. of the Int. Conf. on Data and Knowledge Systems for Manufacturing and Engineering, Hartford, 1987, pp. 65-74.

Care86 Carey, M.J., et. al.: The Architecture of the EXODUS Extensible DBMS, Proc. Int. Workshop on Object-Oriented Database Systems, Asilomar, 1986, pp. 52-65.

Chou86 Chou, H.-T., Kim, W: A Unifying Framework for Versions in a CAD Environment, Proc. of the 12th Int. Conf. on VLDB, Kyoto, Japan, 1986, 1986, pp. 336-344.

Dada86 Dadam, P., et. al.: A DBMS Prototype to Support Exended NF^2-Relations: An Integrated View on Flat Tables and Hierarchies, Proc. 1986 SIGMOD Conf., Washington, D.C., 1986, pp. 356-367.

Dada88 Dadam, P., et. al.: Managing Complex Objects in R^2D^2, Krüger, G., Müller, G. (ed.): HECTOR: Heterogeneous Computers Together, Vol. II (Basic Projects), 1988, pp. 304-331.

Davi78 Davies, C.T: Data Processing Spheres of Control, IBM Systems Journal, Vol. 17, No. 2, 1978, pp. 179-198.

Depp87 Deppisch, U., Obermeit, V.: Tight Database Cooperation in a Server-Workstation Enviroment, Proc. 7th It. Conf. of Distributed Computing Systems, Berlin, 1987, Ed.: Popescu-Zeletin, Le Lann, Kim, IEEE Computer Society Press, Washington, D.C., pp. 416-423.

Ditt87 Dittrich, K.R: Controlled Cooperation in Engineering Database Systems, Proc. IEEE Conf. on Data Engineering (DE-3), Los Angeles, Feb. 1987.

Ditt88 Dittrich, K.R., Lorie, R.A: Version Support for Engineering Database Systems, IEEE Transactions on Software Engineering, Vol. 14, No. 4, Apr. 1988, pp. 429-437.

Fisc89 Fischer, G.: Communication Requirements for Cooperative Problem Solving Systems, Information Systems, Vol. 15, No.1, pp. 21-36, 1990

Härd83 Härder, T., Reuter, A.: Principles of Transaction-Oriented Database Recovery, ACM Computing Surveys, Vol. 15, No. 4, Dec. 1983, pp. 287-317.

Härd87a Härder, T., Rothermel, K: Concepts for Transaction Recovery in Nested Transactions, Proc. ACM SIGMOD 1987 Annual Conf., San Francisco, May 27-29, Order No. 472870, ACM Press, 1987, pp. 239-248.

Härd87b Härder, T., Meyer-Wegener, K., Mitschang, B., Sikeler, A: PRIMA - A DBMS Prototype Supporting Engineering Applications, Proc. of the 13th Int. Conf. on VLDB, Brighton, 1987, pp. 433-442.

Härd87c Härder, T., Rothermel, K: Concurrency Control Issues in Nested Transactions, IBM Almaden Research Center, San Jose, California, Computer Science Department, Aug. 1987, 24 pp.

Härd88a Härder, T: Overview of the PRIMA Project, (Hrsg: Härder, T.), The PRIMA Project - Design and Implementation of a Non-Standard Database System, University Kaiserslautern, SFB 124, Report No. 26/88, Mar. 1988, pp. 1-12.

Härd88b Härder, T., Hübel, Ch., Meyer-Wegener, K., Mitschang, B: Processing and Transaction Concepts for Cooperation of Engineering Workstations and a Database Server, Data & Knowledge Engineering, Vol. 3, 1988, pp. 87-107.

Härd89 Härder, T: Non-Standard DBMS for Support of Emerging Applications - Requirement Analysis and Architectural Concepts, (Ed.: Shriver, B.D.), Proc. of the 22nd Hawaii Int. Conf. on System Sciences (HICSS-22), Kailua-Kona, Hawaii, Volume II, Jan. 1989, pp. 549-548.

Hask82 Haskin, R.-L., Lorie, R.A: On Extending the Functions of a Relational Database System, Proc. of the ACM SIGMOD Conf., Orlando, Florida, 1982, pp. 207-212.

Katz84 Katz, R.H., Lehman, T: Database Support for Versions and Alternatives of Large Design Files, IEEE Transactions on Software Engineering, Vol.SE-10, No.2, Mar. 1984, pp. 191-200.

Katz86 Katz, R.H., Anwarrudin, M., Chang, E: A Version Server for Computer-Aided Design Data, Proc. of the ACM/IEEE 23rd Int. Conf. on Design Automation, Las Vegas, 1986, pp. 27-33.

Katz86 Katz, R.H., Chang, E., Bhateja, R: Version Modelling Concepts for Computer-Aided Design Databases, (Ed.: Zaniolo, C.), Proc. SIGMOD Int. Conf. on Management of Data (Washington, D.C., May 28-30, 1986), ACM SIGMOD Record, Vol. 15, No. 2, June 1986, pp. 379-386.

Kelt87 Kelter, U: Concurrency Control for Design Objects with Versions in CAD Databases, Information Systems, Vol. 12, No. 2, 1987, pp. 137-143.

Kim84 Kim, W., Lorie, R.A., McNabb, D., Plouffe, W: A Transaction Mechanism for Engineering Design Databases, Proc. of the 10th Int. Conf. on VLDB, Singapore, Aug. 1984, pp. 355-362.

Klah85 Klahold, P., Schlageter, G., Unland, R., Wilkes, W: A Transaction Model Supporting Complex Applications in Integrated Information Systems, Proc. ACM SIGMOD, Austin, Texas, 1985, pp. 388-401.

Klah86 Klahold, P., Schlageter, G., Wilkes, W: A General Model for Version Management in Databases, Proc. 12th Int. Conf. on VLDB, Kyoto, Japan, Aug. 1986, pp. 319-327.

Kort87 Korth, H.F., Kim, W., Bancilhon, F: On Long-Duration CAD Transactions, University of Texas, Austin, Department of Computer Science, 1987, 26 pp.

Lori83 Lorie, R.A., Plouffe, W: Complex Objects and Their Use in Design Transactions, Proc. of the Annual Meeting Database Week: Engineering Design Applications (IEEE), may 1983, pp. 115-121.

Lori84 Lorie, R.A., et al.: Supporting Complex Objects in a Relational System for Engineering Databases, Query Processing in Database Systems, Springer Verlag, 1984.

Lian90 Liang, L. Chanson, S.T., Neufeld, G.W.: Process Groups and Group Communications: Classifications and Requirements, Computer Vol. 23, No. 4

Mits88 Mitschang, B: Towards a unified view of design data and knowledge representation, (Ed.: Kerschberg, L.), Proc. of the 2nd Int. Conf. on Expert Database Systems, Apr. 1988, pp. 33-50.

Mits88 Mitschang, B: The Molecule-Atom Data Model, (Ed.: Härder, T.), The PRIMA Project - Design and Implementation of a Non-Standard Database System, University Kaiserslautern, SFB 124, Report No. 26/88, Mar. 1988, pp. 13-36.

Nodi90a Nodine, M.H., Fernandez, M.F., Skarra, A.H., Zdonik, S.B.: Cooperative Transaction Hierarchies, Technical Report No. CS-90-03, Feb. 90.

Nodi90b Nodine, M.H., Zdonik, S.B.: Cooperative Transaction Hierarchies: A Transaction Model to Support Design Applications, Proc. of the 16th Int. Conf. on VLDB, Brisbane , Aug. 1990, pp. 83-94.

Paul87 Paul, H.B., et. al.: Architecture and Implementatin of the Darmstadt Database Kernel System, Proc. 1987 SIGMOD Conf., San Francisco, 1987, pp. 196-207.

Ranf89 Ranft, M.A., Rehm, S., Dittrich, K.R: How to Share Work on Shared Objects in Design Databases, Forschungszentrum Informatik, 1989, 15 pp.

Rehm88 Rehm, S., Raupp, T., Ranft, M.A., Längle, R., Härtig, M., Gotthard, W., Dittrich, K.R., Abramowicz, K: Support for Design Processes in a Structurally Object-Oriented Database System, (Ed.: Dittrich, K.R.), Advances in Object-Oriented Database Systems, Springer-Verlag, Lecture Notes in Computer Science, Vol. 334, 1988, pp. 80-97.

Sidl80 Sidle, T.W.: Weakness of Commercial Database Management Systems in Engineering Applications, Proc. 17th Design Automation Conf., Minneapolis, 1980, pp. 57-61.

Skar90 Skarra, A.: Localixed Correctness Specificaitns for Cooperating Transactions in an Object-Oriented Database, to appear Office Knowledge Engineering, Vol. 4, 1990.

Vanh90 Vanhoedenaghe, M.: A Cooperative Transaction Model, Proc. of the 23trd Hawai Int. Conf. on System Science, 1990, Vol. II HICSS-23.

Weik84 Weikum, G., Schek, H.-J: Architectural Issues of Transaction Management in Multi-Layered Systems, Proc. of the 10th Int. Conf. on VLDB, Singapore , Aug. 1984, pp. 454-465.

Weik86 Weikum, G: A Theoretical Foundation of Multi-Level Concurrency Control, Proc. 5th ACM Symposium on Principles of Database Systems (Cambridge, Mass., 24.03.1986), 1986, pp. 31-42.

Weik87 Weikum, G: Enhancing Concurrency in Layered Systems, Proc. 2nd Int. Workshop on High Performance Transaction Systems, Asilomar, 1987.