The C^3-Locking-Protocol - A Concurrency Control Mechanism For Design Environments

Norbert Ritter
Department of Computer Science, University of Kaiserslautern
P.O. Box 3049, 67653 Kaiserslautern, Germany
e-mail: ritter@informatik.uni-kl.de

Abstract

The CONCORD (CONtrolling COopeRation in Design environments) approach is a processing model for cooperative design processes. It supports the different types of activities predominating in design (transactions, design flows, cooperative actions) by providing a layered activity model allowing to pre-specify design flows and wanted patterns of cooperation and to control the corresponding actions in order to enforce those specifications. CONCORD exploits conventional transactions as elementary building blocks. Although we claim that conventional transactions are needed as elementary building blocks also in cooperative design processes, we think that the corresponding concurrency control mechanism has to be adapted to special requirements of design processes. These requirements are implied by the necessity of interplay between cooperation control, design-flow management, design-tool executions, transaction management, and version management. In this paper, we introduce a locking protocol meeting these requirements. The C^3-protocol controls concurrent manipulations of versioned data and provides an adequate interplay with the higher processing structures (design flow, cooperation). Furthermore, it reflects the semantics of the version model, it avoids blocking of transactions and it takes cooperative design data exchange into account.

Keywords: DBMS, Versioning, Activity Management, Transactions, Design Flow, Cooperation.

1. Introduction

The term *groupware systems* [11] usually refers to a huge class of computer systems which cannot be rigidly distinguished from other classes of computer systems. In [7] groupware is defined as: "*Computer-based systems that support groups of people engaged in a common task (or goal) and that provide an interface to a shared environment.*"

The database research community is currently examining various possibilities of supporting cooperative applications by corresponding components of a database management system (DBMS), e. g. data modeling support (versioning), activity support (workflow, cooperative transactions) or distribution/interoperability support [26]. In this paper, we want to consider the areas of versioning and activity support. Pertaining activity support, database research produced lots of cooperative transaction models [8]. Most of these models are relaxing serializability [13, 10] as underlying notion of correctness (e.g. [1]); others allow the specification of completely user-defined correctness criteria [21]. Although these approaches are well founded and very promising, we think that they are not directly applicable in DB-based design environments, since they do not sufficiently reflect the principles of design work. Within a design environment (or more generally CAD framework [12]), a number of tools is provided, the cooperating users may invoke to manipulate the design data managed by the design data repository [2]. These users are associated with certain design tasks they have to fulfill, and their work is usually bound to certain design methodologies. Thus, we find the following principles:

- design environments are tool-based; designers execute tools to manipulate design data and to create the design objects fulfilling the requirements associated with their (local) design tasks;
- tools are often interactive so that there is a direct communication between designer and tool;

• tool executions typically have a long system residence time;

• tool executions have to fit into the chosen design methodology reflected by certain workflow specifications [9, 27, 5, 19] (design-flow specifications [25, 6]);

• cooperation among designers consists of exchanging preliminary design data; but only meaningful design data (w.r.t. to the design goals) is interesting for cooperation; this means that only results of design phases or results of design tools having a certain design quality are adequate portions of cooperative data exchange;

• design data is usually complex structured and versioned [23], reflecting the principle of stepwise improvement and supporting the possibility to step back to a previous state in order to try a different way to fulfil the design goal; therefore, the data model has to provide explicit and apparent versions.

These principles are reflected by the CONCORD processing model for cooperative design applications [22] and the underlying object and version data model OVDM [17]. CONCORD is a layered approach supporting the different types of activities inherent to design applications (transactions, tool executions, design flows, cooperative actions) at separate layers. As already argued in [22] we claim that cooperative design applications require ACID[1] transactions as elementary building blocks due to consistency. Long ACID transactions are best suited for the implementation of design tools. Regarding the long duration of the transactions to be synchronized, the underlying explicit notion of versioning and the wanted interplay between transaction processing and design-flow/cooperation control, an adequate concurrency control mechanism has to meet the following requirements:

• blocking long transactions is not acceptable, neither is an abort in a late phase of a long transaction;

• the concurrency control mechanism has to take into account the cooperation relationships between different design tasks, the transactions to be synchronized are associated with;

• the mechanism has to reflect the semantics of the data model (versioning principles) and has to take the inherent characteristics of data access into account.

1. ACID stands for atomicity, consistency, isolation and durability and describes the properties of conventional database transactions [13, 10].

We will argue that a pessimistic locking protocol is best suited in our scenario and will introduce the C^3-protocol (Concurrency Control in CONCORD) used within the CONCORD approach to isolate tool executions. Additionally, the C^3-protocol is able to reflect the specific requirements of the surrounding cooperative environment.

Overview of the paper

After this short introduction, we will first introduce the major concepts of the object and version model OVDM (Sect. 2) used in our design environment to model and represent design data. Sect. 2 will also discuss the principles of the corresponding language OQL (object query language). We will see that these principles restrict the possible techniques for concurrency control. Since, due to space restrictions, we cannot detail the CONCORD model, we will introduce only those aspects relevant for concurrency control (Sect. 3) and will, afterwards (Sect. 4), give a detailed discussion of the C^3 mechanism. Sect. 5 will give a conclusion.

2. Versioning Concepts

2.1 Basic Concepts

In [17] the object and version data model (OVDM) is proposed for the management of design data. Since the operations issued by designers to manipulate versioned data have to be synchronized, we will outline the major concepts of this model, illustrated in Fig. 1.

Complex Objects and Versions

Complex Objects are identifiable occurrences of complex object types and are given as structured sets of elementary data. In its totality a complex object is described by object attributes. An (complex) object combines *elementary objects*, which can be compared with the tuples of the relational model. In addition to the features of the relational

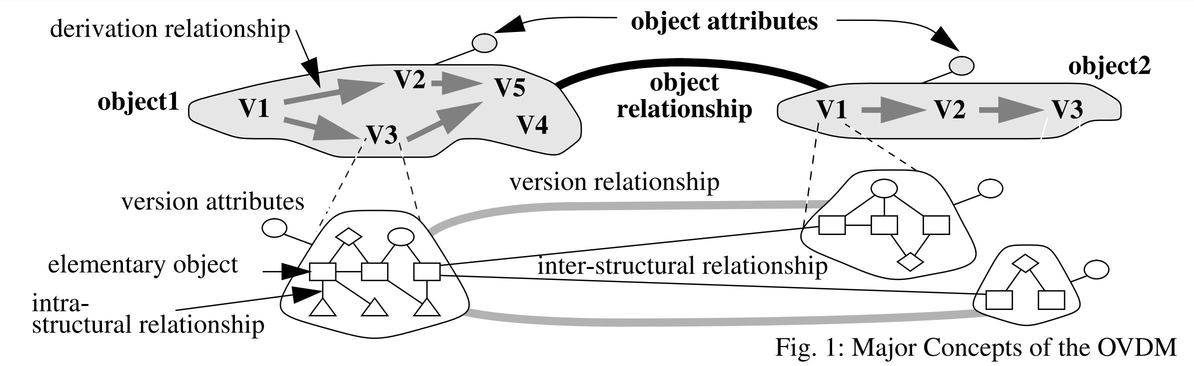

Fig. 1: Major Concepts of the OVDM

model, elementary objects can be connected via (typed) *structural relationships*. Obviously, it is one of the major tasks of a design process to create "contents" of objects, i.e., nets of elementary objects. Design is an iterative process, which typically leads to several (similar) nets of elementary objects, which we call *versions*. Thus, versions are different states of the net of elementary objects constituting a complex object. In this way, versions are capturing the various object states derived during the design process with the intention of a stepwise improvement of preliminary data in order to reach the (partial) design goal. The relationships between the versions of a single object, representing the derivation of new versions out of existing versions, are managed in form of a derivation graph, which, in turn, can be organized as list, tree or acyclic graph.

Relationships between Objects / Versions

Objects can be connected by object relationships. Obviously, these relationships must also be captured at the version level. Therefore, version relationships can be seen as refinements of object relationships. Version relationships depend on the existence of relationships between the corresponding objects. In addition to the explicitly represented version relationships implicit relationships can be modeled resulting from data overlapping. Overlapping can occur directly by shared elementary objects or by inter-structural relationships which should only be interpreted in configurations for consistency reasons. In Fig. 1 we see that version 3 of object 1 is connected with versions 1 and 3 of object 2 by version relationships. Suppose that every version is representing its object completely. This implies that version 3 of object 1 actually can only be in relationship with either version 1 or version 3 of object 2. Such (semantic) integrity constraints are to be checked at the level of configurations.

Configurations

It is the goal of configuring to establish consistent units by selecting certain versions out of the set of versions stored in the database [15]. Configurations (not shown in Fig. 1) are occurrences of specified configuration types which are usually associated with special integrity constraints expressing requirements the corresponding unit of versions has to fulfill. Configuration types are defined on the basis of object types and relationship types. A configuration type is given as a tree where the nodes represent complex object types and edges are standing for paths which, in turn, may consist of complex object types and relationship types. The configuring process starts with a version of the root type. For each inner node, a version selection must be done satisfying certain requirements. For example, it can be enforced that at a certain node all selected versions must belong to distinct objects or that certain semantic integrity constraints must be fulfilled on the selected versions or the resulting structure, respectively. Furthermore, an important property of configuration types is their structure, i.e., hierarchical, or even recursive.

Example

Since we cannot explain all OVDM concepts in a detailed manner, we want to illustrate the major concepts with an example from the area of VLSI design. Chip planning [29] is a very important phase within the VLSI design process. Here, a hierarchy of VLSI cells is considered, reflecting a decomposition of the resulting chip's functionality. In the following, we will refer to the entire set

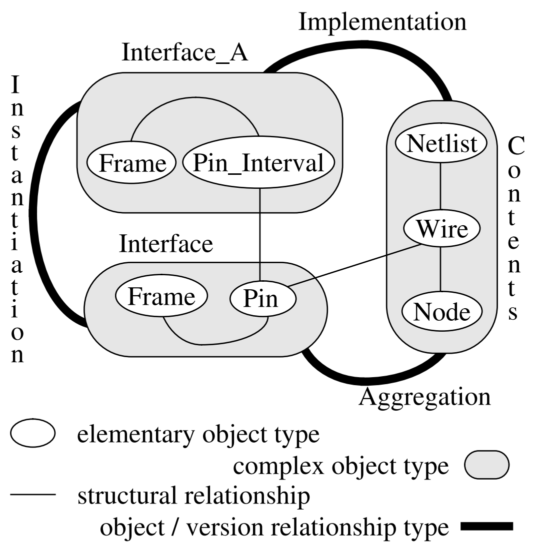

Fig. 2: Applying OVDM Concepts

```
DEFINE OBJECT_TYPE Interface_A
   COMPONENTS      (  Frame, Pin_Interval    )
   OBJECT_ATTRS    (  ifa_id:              IDENTIFIER,
                      name:                STRING,
                      function:            STRING     )
   VERSION_ATTRS   (  ifa_v_id:            IDENTIFIER,
                      scale_factor:        REAL,
                      no_feedthroughs:     INTEGER  )
   VERSION DERIVATION GRAPH IS TREE;

DEFINE LINK_TYPE Aggregation
   FROM  Contents TO Interface: (1,1000)
   FOR VERSIONS: (1,1000),
   FROM  Interface TO Contents: (1,1)
   FOR VERSIONS: (1,*);

DEFINE CONFIGURATION_TYPE Cell FOR Interface
   ATTRS   (  cell_id:         IDENTIFIER,
              description:     STRING    )
   VIA (  Interface-<Instantiation>Interface_A-<Implementation>
          Contents-<Aggregation>Interface OBJECT_SEL )
   FOR    Interface USE Cell;
```

of tools applied during the chip planning phase as the *chip planner*. A single chip-planner run considers a single cell of the mentioned hierarchy together with its direct subcells. As a result, the chip planner delivers an arrangement of the subcells within the area given for the supercell (topography). Modeling the design data by means of OVDM concepts may lead to structures illustrated in Fig. 2.

On the left-hand side the object structures are depicted; the right-hand side shows sample data definition statements (due to space restrictions, we just give a single example for the definition of a versioned complex object type, an object/version link type and a configuration type, respectively). The graphical illustration of the schema shows three complex object types. Interface and Interface_A describe the interface of a VLSI cell. Whereas Interface represents a concrete interface, and, therefore, contains the elementary object types Frame and Pin, Interface_A is an abstraction of the interface description (elementary object type Pin_Interval), so that several Interface objects can be associated with the same Interface_A object. The complex object type Contents consisting of the elementary object types Netlist, Wire and Node describes the internal construction of a cell. Considering the (intuitive) semantics of the relationship types Instantiation, Implementation and Aggregation, it becomes obvious that the path Interface-Interface_A-Contents connects a supercell with its direct subcells. Presupposing, the three complex object types to be versioned, we have to configure a cell by, firstly, selecting a version of this cell and, secondly, selecting fitting versions of the subcells. For that reason, the configuration type Cell is defined. The path definition in the VIA clause reflects the above mentioned connection of a cell with its direct subcells. A configuration only contains versions belonging to object types opening or closing a path definition in the corresponding configuration type. More information about the data definition language is given in the following subsection.

2.2 Object Query Language

The example given in the previous subsection contains a sample data definition statement for each, a complex object type, an object/version link type and a configuration type. The elementary object type definitions were omitted, because they are very similar to the object definition in the relational data model. In this subsection, we will briefly discuss the object query language (short OQL) being the language of the OVDM.

Data Definition

The definition of a complex object type (see Fig. 2) contains clauses for associating elementary object types (COMPONENTS clause), specifying object attributes and version attributes as well as defining the structure of the derivation graph. The sample definition of link type Aggregation shows that different cardinality restrictions can be specified at object level and version level. The configuration type Cell (see Fig. 2), in turn, enforces to select exactly one version per subcell interface. This is expressed in the corresponding path definition (VIA clause). In general, a configuration type may contain several (hierarchically structured) path definitions. The Cell configuration type contains just one. Starting with an Interface version (supercell) the defined path is considered leading to a number of interface versions (subcells). Now, the OBJECT_SEL specification enforces that only one version of a single object can be incorporated into one configuration. Further possibilities are, firstly, to incorporate all reached subcells (NO_SEL), or, secondly, to let the user arbitrarily select out of the set of reached subcells (VERSION_SEL). Further requirements, versions have to fulfill in order to be acceptable for incorporation into a configuration can be expressed by means of semantic integrity constraints.

Data Manipulation

The data manipulation part of the OQL contains statements for the creation and deletion of complex objects, versions, links, configurations as well as for the modification (update in place) of objects and versions. Due to space restrictions we just give examples (see Fig. 3, left-hand side). The first statement creates an Interface_A object and the second a corresponding version.

Data Selection

The select statement provides means for retrieving (sets of) complex structures and configurations. Complex structures consist of object/version nodes and relationships. On the right-hand side of Fig. 3 some examples are shown. The first statement retrieves object level information (object attributes, object relationships), i.e., the nodes of the resulting complex structures are object nodes. The FROM clause expresses that simple hierarchies are considered consisting of a Contents object and all Interface objects used 'within this Contents'. In the same way, complex structures can be defined in the 'select version' statement retrieving version level information (version attributes, version relationships, elementary objects, structural relationships). In general, the FROM clause allows the specification of nets, but with the restriction that a unique root is specified. Thus, a select statement retrieves for each root object/version a complex structure as defined in the corresponding FROM clause. OQL statements may contain very powerful predicates (WHERE clause) expressing value-based as well a structural requirements. Due to space restrictions, we cannot detail this matter.

2.3 Application Programming Interface

OQL can be used as ad-hoc query facility as well as embedded into a higher programming language (C^{++}). The corresponding processing concept will be discussed in detail in [20]. The major principle of this processing concept is to distinguish three phases. First, select statements can be issued to perform checkout operations transmitting the resulting complex structures from the server to an object buffer residing on the wokstation. Second, a processing phase is supported by a hierarchical cursor concept allowing to traverse and manipulate data delivered by the checkout operation. The corresponding run time system operators reflect the semantics of the OQL. Third, a checkin operation propagates the changed information (net effect) back to the design-data management system (design data repository) residing on the server. The mentioned API provides means for the integration of design tools into the design environment.

3. Effects of Cooperation and Design Flow

After having discussed the major principles of the underlying object and version data model, we want to identify those aspects of cooperation control and design-flow management influencing concurrency control. As already mentioned in the introduction, the interplay of transaction management, design-flow management, and cooperation control is a crucial issue. Since we cannot introduce complete models for design-flow management and cooperation control, we, again, have to refer to [22, 13] discussing the CONCORD pro-

```
CREATE OBJECT
  ( name := "1-Bit-Adder",
    function := "s, c := a + b")
FROM Interface_A;

CREATE VERSION
  ( scale_factor := 1.0,
    no_feedthroughs := 0)
  COMPONENTS
  Frame          ( f_id = 1234                        )
  Pin_Interval   ( pi_id = 5678                       )
                 ( name := "x",
                   position := [(0,1,4), (0,1,7)]     )
FROM Interface_A
WHERE ifa_id = 4711;
```

```
SELECT  OBJECT ALL
FROM    Contents-<Aggregation>Interface
WHERE   EXISTS Interface: name = "1-Bit-Adder";

SELECT  VERSION ALL
FROM    Instance-<Instantiation>Interface_A-
        <Implementation>Contents
WHERE   EXISTS Interface_A: ifa_id = 4711;

SELECT  CONFIGURATION
FROM    Cell
WHERE   cell_id = 0815;
```

Fig. 3: Sample OQL Statements

cessing model. CONCORD considers transaction management, design-flow management and cooperation control at separate system layers. Since there is the requirement of an adequate interplay among these layers, we want to outline the implications for concurrency control preparing the discussion of the concurrency control mechanism in Sect. 4.

3.1 Cooperation Control Principles

At its highest layer (cooperation control) CONCORD provides the design activity (DA) concept. A design activity is the operational unit representing a design task. During the design process a hierarchy of concurrently active DAs may be dynamically established resembling a hierarchy of concurrently active tasks. Among DAs different types of cooperation are modeled by means of different cooperation relationship types. Besides delegation and negotiation relationships usage relationships are supported enforcing certain protocols for the cooperative exchange of (preliminary) design data between DAs. In this paper, we will only consider usage relationships, because they influence concurrency control. Three types of cooperation protocols can be associated with usage relationships: (1) implicit; (2) explicit and dynamic; (3) explicit and pre-planned. These cooperation types are explained in [13]. Since in CONCORD implicit cooperation is modeled at the level of tool executions and not at the level of database operations, only explicit cooperation directly affects concurrency control.

A prerequisite of *explicit cooperation* is that each DA is (logically) associated with a local object pool. This data may, first of all, only be manipulated by design-tool applications issued by this DA. By using explicit cooperation operations (e.g. transmitting objects/versions from one DA to another one, or allowing a cooperating DA to access own data in a certain mode which can be 'read', 'derivation', or 'update') the set of data accessible for a DA may be extended dynamically (w.r.t. the design phases of the cooperating DAs).

DAs may be allowed to dynamically issue cooperation operations in certain design phases (*dynamic cooperation*) or the execution of cooperation operations may be pre-planned. The latter means that the design-flow description of a DA, which is specified in advance, does not exclusively contain design-tool invocations but also cooperation operations (*pre-planned cooperation*).

3.2 Design-flow Management Principles

With each DA a responsible designer is associated who is in charge of the execution of a design flow having the goal of the associated DA in mind. The design-flow system has to accomplish the task of *guiding the user (designer) through the complex activity*. The challenge here is to be able to pre-plan and schedule a complex activity *without* restricting the cooperating user's creativity urgently needed to capture the inherent complexity in cooperative problem solving. Design flows can be specified in advance and rely on the concepts of *design states* (sets of current design objects) and *transitions* (actions, usually design-tool executions) making the DA to change its state.

Pertaining control flow and data flow the CONCORD design-flow model is comparable to well

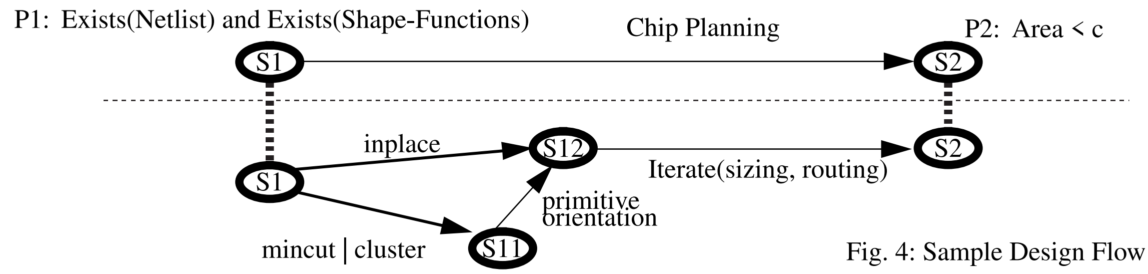

Fig. 4: Sample Design Flow

known workflow models [9]. Due to space restrictions, we cannot discuss this matter and have to refer to [24]. We just want to give an abstract example. Fig. 4 shows at its upper part a single transition consisting of a pre-condition (P1), a post-condition (P2) as well as an abstract activity specification (Chip Planning). Under certain circumstances (e.g. well experienced designer) such a high-level specification is sufficient and leaves the designer a maximum degree of freedom. Thus, he can dynamically decide which tools to apply in order to get to the state S2. In other cases, it may be adequate to give a more detailed specification including concrete tool invocations (see lower part in Fig. 4)[1].

At run time, the design-flow management system controls the execution of the specified design flow, if necessary in interaction with the responsible designer. Whenever the design-flow management system does not have complete information to decide about the next step, the designer is asked to provide the missing information.

Due to simplicity of explanations, we associate a single long transaction with a single design-tool execution. In the following, we will use the term DOP as synonym standing for *design operation* and abstracting from a concrete design tool. Although a DOP is a long transaction, it has the ACID properties of conventional transactions.

3.3 Implications for Concurrency Control

Considering cooperation control, design-flow management and design-tool executions, we want to recall the following characteristics which influenced the design of the concurrency control mechanism which will be discussed in Sect. 4:

1. Data and control flow details as well as pre- and post-conditions have been omitted in the lower part of Fig. 4).

- DAs are cooperative units, each associated with a designer who is in charge for the DA-internal design-flow processing;
- design objects (versions, objects, configurations) are primarily manipulated by the DA they are owned by; nevertheless, access rights (‘read’, ‘derivation’, ‘update’) may be granted to other DAs along cooperation relationships;
- the design-flow management system supports the designer in fulfilling the design task; whenever the design-flow management system does not have complete information to decide about the next step, designer interaction is triggered;
- design tools are long running activities which are to be executed in isolation; tools are often interactive and, since data flow can be specified at the design-flow level, the processing context is usually known before starting execution.

4. The C^3-Locking-Protocol

After having introduced the major principles of the OVDM and the major processing structures, we now want to discuss the concurrency control mechanism we use to synchronize DOPs. A major benefit of versions is that a higher concurrency of processing is possible; thus, the protection realized by a concurrency control mechanism should not invalidate this advantage by inadequate protection mechanisms. For example, during the derivation of a successor version the considered version must be protected sufficiently, but it is not necessary to remove it completely from other transactions’ reach. W.r.t. the processing structures outlined in the previous section, a concurrency control mechanism faces the following additional requirement: Since the (logically) local databases of the DAs involved in the design process usually overlap due to cooperation and this constellation may change dynamically, it does not make any sense to partition the database and let the DOPs of a certain DA only access the local database associated with this DA. It is more adequate to manage the design data in a central data repository and to consider it as additional task of the synchronization mechanism, to decide whether or not the DAs (the DOPs to be synchronized are associated with) may access requested data.

Looking for an adequate concurrency control mechanism we, firstly, had to decide about using an optimistic approach [4, 16], a timestamp mechanism [4], or a pessimistic protocol [3, 10]. In conflict situations, optimistic and timestamp mechanisms have high aborting rates. Optimistic approaches detect conflicts very late (evaluation phase after the processing phase) and resolve these conflicts by aborts. This is not acceptable for long running transactions. Timestamp approaches put long transactions at a disadvantage, too (the longer a transactions runs, the higher is the possibility that it is ‘too late’ to access certain data). For these reasons, we decided to consider locking protocols. Here, our first attempt in exploiting the elegance of a granular lock protocol (GLP) [10] failed, because of inherent properties of OVDM/OQL: The database is accessed by select statements (see Sect. 2). Considering the result sets of select statements, we realize that the resulting structures are highly meshed, heterogeneous, complex structures. This prevented us from finding natural hierarchical levels, which would allow to exploit the commonly known advantages of implicit locks (as used in granular lock protocols). Therefore, we found ourselves forced to consider R/X based lock protocols. The results of these examinations will be discussed in the remainder of this section. An adequate locking protocol for our scenario has to meet the following requirements:

- Since the OVDM considers complex objects as sets of explicit versions and the OQL does (naturally) also reflect these semantics and provides statements at the object level as well as the version level, adequate lock modes for the different kinds of manipulations must be found.
- Especially, the derivation of new versions must be protected by sufficient modes, but compatibilities of locks must also leave as much concurrency as possible so that the natural advantages of versions are not invalidated.
- Usually, DOPs take a long execution time. Conventional locking techniques enforcing serializability [14], e.g. the two-phase locking protocol [10], cannot be applied (at least not in their pure form), because these protocols apply blocking: a transaction which wants to access data which is currently locked by a concurrent transaction in a conflicting mode must (synchronously) wait until the lock holding transaction commits. This may lead to unacceptable long waiting situations. Therefore, at least the way of acquiring locks must be adapted.

The first of the three mentioned points refers to the question of which lock modes are to be provided; this question will be answered in Sect. 4.1. The second point asks about the compatibilities between the modes; this matter will be discussed in Sect. 4.2. The third point refers to the rules which must be observed during the locking protocol; this will be outlined in Sect. 4.3.

4.1 Lock Modes

In this section, we want to consider the lock modes needed for the manipulation of objects and versions. We will start discussing the lock modes needed to protect version manipulations.

The discussion of the version lock modes is summarized in the table below containing for each mode its internal effect (rights of the lock-holding transaction) and its external effect (remaining rights for others).

As already mentioned, we had to introduce shared and exclusive locks for versions. The VS mode corresponds to the well known read mode and VX corresponds to the exclusive mode known from simple R/X protocols. One of our major goals for tailoring the lock modes to versioning was to protect version derivations adequately. Therefore, we introduced the VD mode allowing concurrent version derivations. The necessity of introducing different shared modes (VS and VD) becomes clear by the discussion of the third shared mode:

VDX. The VDX mode realizes an exclusive derivation lock so that only the lock holding transaction is allowed to derive a successor, others may just read the version. This restriction is necessary in one of the following two situations:

• The structure of the derivation graph does not allow more than one successor for the considered version. For example, a list structure just allows deriving one successor for the head node. Thus, it does not make any sense to give more than one transaction the derivation right.

• The application restricted the proliferation of the derivation graph in a way that does not allow to derive multiple successors for the considered version. Generally, there are the following possibilities of dynamically restricting the growth of a derivation graph:
one derivation at a time: Even in the case that the structure of the derivation graph would allow multiple successors, the application may restrict the derivation of successors for a certain version by allowing only a single derivation at a time.
derivation ban: In contrast to the first case, further derivations can be utterly forbidden.
no new roots: Besides the restriction of the number of roots in the 'define object type' statement the possibility of inserting new root versions can be dynamically switched off.

Coming back to our discussion of the shared modes supported at the version level (VS, VD, VDX) it now becomes clear that we have to distinguish VD and VDX modes: Whenever a transaction wants to derive a successor of a certain version the current derivation graph structure (especially the currently possible derivations for the considered version) determines, whether VD or VDX must be acquired.

Next, we have to explain the VXC mode being a very special lock mode. It allows to handle the situation described in the following. Suppose, a transaction applies for getting exclusive access to a version and this access can currently not be granted because of conflicts with concurrent transactions having access (in a shared mode) to this version. This situation can be handled by maintaining the request in the following way. The transaction gets read access (note, this is compatible with locks of concurrent transactions) and as soon as the VXC lock is acquired no further locks on the considered version can be acquired. As soon as all conflicting locks are released (commit of the corresponding transactions) the read lock is upward converted to an exclusive lock and the transaction may update the version. Handling this situation in the mentioned way, is especially reasonable in our checkout/checkin scenario. Basically, a transaction's checkout operation merely needs to acquire a read lock. The subsequent manipulations are performed on copies saved in the object buffer on the workstation. At the latest, the propagation of the changed information (checkin operation) requires the exclusive lock. If there is a good probability that the initially conflicting transactions will be out of the system until the requesting transaction issues its ckeckin, the usage of the VXC lock mode can be recommended. The probability can be estimated by heuristics and sta-

Version Lock Mode	Internal Effect	External Effect
VS (version read)	reading the version;	reading the version, deriving a successor version;
VD (version derive)	reading the version, deriving a successor version;	reading the version, deriving a successor version;
VDX (version exclusive derive)	reading the version, deriving a successor version;	reading the version;
VX (version exclusive)	reading the version, deriving a successor version, updating the version;	no rights;
VXC (version exclusive convertible)	reading the version;	no (further) rights;

tistical information, e.g. the average run-time of the corresponding design tools.

After having discussed the version lock modes we now want to switch to the object level. The OVDM comprehends objects as sets of explicit versions. Therefore, on one hand, manipulating an object means manipulating the entire corresponding set of versions. On the other hand, manipulating a single version of an object leaves the possibility of 'sharing' this object. This idea is reflected by the modes discussed below. Again, a table gives an overview of the internal and external effect of each of the object lock modes.

At the object level, we also distinguish shared modes (OS, ODX) from exclusive modes. Updating the object, e. g. explicitly modifying the derivation graph, requires an exclusive lock. The lock mode OXC has the same semantics at the object level as the VXC lock mode at the version level. The ODX lock mode is a very special mode which we had to introduce to handle the following special case. There are mechanisms to restrict the growth of the derivation graph of an object. For example, a list structure can be enforced by the derivation graph clause of the object-definition statement. This implies that after having created an empty object (no versions), it has to be ensured that only one root version can be inserted. Therefore, the corresponding object is locked in ODX mode implying that concurrent transactions must not insert a second root version.

4.2 Compatibilities

After having introduced the basic lock modes for objects and versions in the previous subsection, we will now consider the corresponding compatibilities. The tables at the right-hand side of Fig. 5 illustrate the compatibilities. One more time we want to consider the version level and the object level independently. The dependencies between locking objects and locking versions will be discussed in the following subsection.

Let us first consider the upper table (compatibilities at the version level). Most of the compatibilities concerning shared locks are very intuitive. The two cases marked by an asterisk cannot occur, because the underlying (current) derivation graph structure determines whether a VD or VDX lock must be considered. Furthermore, it is worth mentioning that the matrix is not symmetrical. The reason is the conversion lock implying that as soon as a VXC lock has been acquired no further lock can be given on the considered version.

The object matrix is symmetrical, neither. Again, the reason is given by the meaning of the OXC lock mode.

4.3 The Locking Mechanism

This section will give insights into the C^3 locking procedure. The corresponding algorithm relies on the compatibilities of the different lock modes (see above) which are the foundation of a decision, whether or not a certain lock can be granted. Before discussing the locking procedure in detail, we will firstly introduce some further principles.

Pre-Specification of Access Mode

In engineering applications, the processing context of a design step is usually known in advance. Therefore, we extend select statements by a mode-specification clause, indicating whether the data is needed for update, read or derivation access. Thus, the lock manager can check the com-

Object Lock Mode	Internal Effect	External Effect
OS (object shared)	reading object attributes, version manipulations;	reading object attributes, version manipulations;
ODX (object exclusive derive)	reading object attributes, reading versions, creating a version;	reading object attributes, reading versions, updating versions;
OX (object exclusive)	all rights on the object and its versions;	no rights;
OXC (object exclusive convertible)	reading object attributes, manipulating versions;	no (further) rights;

lock held by a concurrent transaction
lock requested

		VS	VD	VDX	VXC	VX
SELECT VERSION [FOR READ]	VS	+	+	+	-	-
SELECT VERSION FOR DERIVATION	VD	+	+	*	-	-
	VDX	+	*	-	-	-
SELECT VERSION FOR UPDATE	VXC	+	+	+	-	-
	VX	-	-	-	-	-

lock held by a concurrent transaction
lock requested

		OS	ODX	OXC	OX
SELECT OBJECT [FOR READ]	OS	+	+	-	-
SELECT OBJECT FOR DERIVATION	ODX	+	-	-	-
SELECT OBJECT FOR UPDATE	OXC	+	+	-	-
	OX	-	-	-	-

Fig. 5: Compatibilities

patibility of the request with currently running transactions already during the corresponding checkout step. As we will see later, this principle avoids lock upgrades during the checkin steps and, thereby, also avoids unacceptable wait situations. The mentioned mode-specification clause of select statements is also shown in Fig. 5. It illustrates the relationship between mode specification and corresponding lock modes.

Lock Negotiations

In the case that the lock mode implicitly asked for by the mode-specification clause cannot be granted, the system reacts as follows. The lock manager finds out the highest[1] possible lock mode for the requested set of data. For example, if for a set of versions (set of complex structures at the version level) update access has been requested and at least one of these versions is currently in use (shared mode) by a concurrent transaction, the lock manager offers the update-convertible (VXC) mode for that data. The transaction may now chose from the following possibilities:

- accepting the offered mode;
- waiting asynchronously: the application proceeds processing and gets a message as soon as the data can be delivered in the requested mode and can decide again, whether or not it wants to get the data in the originally requested mode;
- waiting synchronously: waiting (and being blocked), until the data can be delivered in the requested mode;
- canceling the request.

1. We suppose the complete order: read < derivation < update-convertible < update

Lock Mode Downgrade

Since we introduced the mode specification in association with set-oriented queries, the situation may occur that the application gets a set of objects/versions in a certain mode and it, actually, does not need this mode on each element of the query's result set. To handle this situation, we support an application-initiated downgrade of locks to read mode.

General Rules

Before going into the locking procedure, we give some general rules, the locking mechanism has to take into account:

- As already mentioned, a select statement delivers a set of complex structures. In the case of a 'select version' statement, the nodes of the complex structures are versions, in the case of a 'select object' statement the nodes are objects. For each node of each complex structure in the result set of the select statement, it has to be checked, whether or not the mode specified in the mode-specification clause can be granted.

- In the case of a ‘select version for derivation’ statement each of these checks follows three steps:
  (1) Do the DA constellation and the corresponding cooperation relationships allow derivation access?
  (2) Must a VD or a VDX lock be acquired?
  (3) Is the determined lock mode compatible to locks held by concurrent transactions?
- In the case of a ‘select object’ statement it is sufficient to check compatibilities at the object level.
- In the case of a ‘select version’ statement for each version in the result set it has to be checked first, whether OS mode can be granted for the corresponding object. Thus, the actual lock modes necessary for version manipulations are pairs (OS/VS, OS/VD, OS/VX, OS/VXC).

Locking Protocol

Now, we can give a high-level description of the Lock_Manager process enforcing the protocol. This process has an interface to the following components: the DOP, the transaction manager handling DOPs, and the deadlock manager maintaining (synchronous and asynchronous) wait-for relationships [10]. Since we do not want to detail the deadlock detection and resolution within this paper, we just want to mention that as soon as a deadlock has been detected that DOP, which entered its manipulation phase at the latest, is aborted. In the following, we will give a high-level description of the lock manager and will explain how it works.

```
PROCESS Lock_Manager
  LOOP
  Accept_Request;
  CASE Request OF
    tm_release[1]:  Release_Locks; EXIT;
    dop_aquire:   Determine_Mode;
      IF possible_mode < wanted_mode THEN
      /* rejected < read < derivation < upd_conv < update */
        Lock_Negotiations;
        CASE negotiation_result OF
          cancel: EXIT;
          accept: Lock_Data_In_Possible_Mode; EXIT;
          asynch_wait,
          synch_wait:
            Pass_Request_To_Deadlock_Manager; EXIT;
        END CASE;
      ELSE Lock_Data_In_Wanted_Mode;
      END IF;
    dop_downgrade:  Check_Corresponding_Locks;
      Downgrade_Locks_To_Read_Mode;
      Notify_Deadlock_Manager;
    dm_lock:  Lock_Data_In_Wanted_Mode; EXIT;
    dm_negotiate:  Lock_Negotiations;
      CASE negotiation_result OF
          cancel: Exit;
          accept: Lock_Data_In_Wanted_Mode; EXIT;
      END CASE;
  END CASE
  END LOOP
END Lock_Manager;
```

1. The case labels at this level represent the requests, which are handled by the lock manager. The first part of the label name stands for the component which initiates the request and the second part of the label name indicates the actual kind of request. Abbreviation tm stands for transaction manager and dm stands for deadlock manager.

Let us consider the lock manager as a process consisting of a single loop where each pass represents the processing of a single request. The labels of the outermost case statement represent the possible requests. The first request we want to discuss is acquiring locks. The application (DOP) indicates within a single checkout step that it wants to acquire locks of a certain mode (mode-specification clause) on a certain set of objects (result set of the corresponding select statement). To handle this request, the lock manager first has to check the compatibility of the wanted locks with current locks. There is a conflict as soon as there is an incompatibility pertaining one object contained in the considered query result set. The compatibility check is done within the function Determine_Mode (see below). In the case of conflicts this function determines the highest, not conflicting mode. Note that the application can only specify the modes ‘read’, ‘derivation’ and ‘update’ in the mode-specification clause of OQL statements. The mode upd_conv which has been introduced above cannot be applied for, it is just a possibility which can be offered by the lock manager if the wanted update mode conflicts. In the case that derivation access is wanted by the application, for each single version in the query result it must, first of all, be determined, whether VD or VDX mode must be checked. As already mentioned, this depends on the derivation graph structure as well as on restrictions given by the application.

Occurring the situation that the mode wanted by the application cannot be granted, the lock manager starts a communication with the DOP (see above: lock negotiations). In the case that the application is not interested in getting the data in a mode lower than the requested one and does not want to wait, the processing of the request is finished by leaving the process loop. In the second case (the application accepts the offered mode), the data is correspondingly locked. In the third case (the application decides to wait synchronously or asynchronously), the lock manager passes the request to the deadlock manager. The deadlock manager, in turn, is expected to pass back the request as soon as the actually wanted mode can be granted[1]. The downgrade request is also issued by the application (DOP), as already mentioned above. The requests dm_lock and dm_negotiate are requests issued by the deadlock manager passing back requests recently issued by the application. They indicate that the recently wanted mode can be granted now. In the case of an asynchronous wait situation (dm_negotiate), the application is asked again, whether or not it still wants to get the data in the acquired mode.

```
PROCEDURE Determine_Mode (ta: ta_id_type;
  sov: set_of_version_id_type;
  wm: lock_mode_type
  /* (read, derivation, update, upd_conv) */;
  VAR sov_vd: set_of_version_id_type;
  VAR sov_vdx: set_of_version_id_type): answer_type
  /* answer_type =
       (rejected, read, derivation, upd_conv, update) */;
BEGIN
  FOR EACH v IN sov DO
    CASE wm:
      read: IF NOT Compatible (ta, v, VS) THEN
              RETURN rejected;
            ENDIF |
      derivation: CASE Restrictions (v)
                    VD: IF Compatible (ta, v, VD) THEN
                      Insert (v, sov_vd);
                      ELSE RETURN
                      Determine_Mode (ta, sov, read);
                      ENDIF |
                    VDX: IF Compatible (ta, v, VDX) THEN
                      Insert (v, sov_vdx);
                      ELSE RETURN
                      Determine_Mode (ta, sov, read);
                      ENDIF |
                    NO_DERIVATION: RETURN rejected;
                  ENDCASE; |
      update:    IF NOT Compatible (ta, v, VX) THEN
                   RETURN
                     Determine_Mode (ta, sov, upd_conv)
                 ENDIF;
      upd_conv:IF NOT Compatible (ta, v, VXC) THEN
                   RETURN
                     Determine_Mode (ta, sov, derivation)
                 ENDIF;
    ENDCASE;
  ENDFOR;
  RETURN wm;
END Determine_Mode;
```

1. Obviously, the maintenance of these waiting situations includes preventing further conflicting locks. That means that the function Compatible (see algorithm Determine_Mode) has to consider explicit locks as well as waiting relationships.

Example

After having discussed the major principles of synchronizing version manipulations, we want to give an illustrating example further motivating and justifying the proposed concepts. Let us consider two design tools from the area of VLSI design. Tool t1 supports interactively changing the positions of subcells within the area assigned to their supercell. Tool t2 is also an interactive tool browsing the Interfaces and allowing the modification of their common Interface_A object/version. Suppose, DOP1 and DOP2 are incarnations of t1 and t2, respectively.

We now want to consider a conflict situation which would occur with the following sequence of events. During its checkout phase DOP1 issues the ‘select version’ statement shown at the left-hand side of Fig. 6. Since the OID of the root version of the complex structure is given, the result contains just one element (also depicted at the left-hand side of Fig. 6). DOP1 gets the complex structure in the wanted derivation mode so that internally the depicted versions (nodes of the complex structure) are locked. We assume that versions I1 and IA1 must be locked in VDX mode, because of the underlying derivation graph structure (or dynamically specified restrictions). Next,

SELECT **VERSION ALL FOR DERIVATION**
FROM Super(Interface)-Interface_A-Contents-Sub(Interface)
WHERE Super. if_id = I1;

SELECT **VERSION ALL FOR DERIVATION**
FROM Interface_A-Interface
WHERE **EXISTS** Interface: if_id = I1;

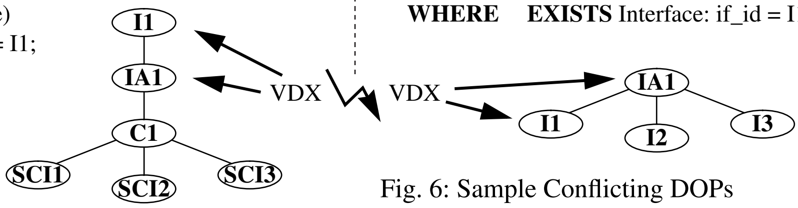

Fig. 6: Sample Conflicting DOPs

DOP2 issues its checkout step (see Fig. 6, right-hand side). Since one of the complex structures contained in the result set contains versions which are already locked by DOP1, the wanted derivation lock cannot be granted. The lock manager notifies DOP2 that currently the highest possible mode is ‘read’ and that statistical and heuristic information indicates that the specified data will probably be free after a certain duration[1]. Now, DOP2 can decide on how to react. Taking this decision can be pre-planned or can be done dynamically by asking the designer who is responsible for the corresponding DA. The second solution seems to be more adequate for an interactive tool. Suppose, the designer decides for ‘wait’, because there is running a second tool within a parallel DOP. Before entering the processing phase, DOP1 downgrades the locks on I1 and IA1, because it is only interested in manipulating the contents information. As soon as this downgrade is executed by the lock manager, DOP2 is provided by the wanted data in the wanted mode.

Deadlocks

It is worth mentioning that deadlocks are really very infrequent in our scenario. A deadlock can only appear, if there are two transactions, both issuing at least two checkout steps and requesting data currently held by the other one, and both deciding to wait for the data being released. Besides that, deadlocks can, usually, be determined and resolved very early (w.r.t. DOP duration). Typically, design tools checkout the needed data (processing context) before entering the processing phase. This matter of fact, together with the above introduced concept of pre-specifying the wanted access mode in the select statement of a checkout step enables the deadlock manager to determine the deadlock very early and to resolve it by issuing negotiations or aborting one of the two transactions. The decision of restarting the corresponding tool is left to the design-flow management system, which, in turn, may ask the designer to take the decision.

Correctness Considerations

The protocol introduced in this section enforces serializability of version manipulations. We do not want to give a formal prove of this statement, since it is very intuitive. Therefore, we just want to give the following explanations. In comparison with the well known case of not-versioned data synchronized by R/X lock techniques [10], our data model additionally provides versions of objects. Our synchronization protocol protects also version manipulations by shared and exclusive locks. Therefore, we have especially to consider the case of version derivations. Reading a version in parallel to derivation (VS compatible to VD; VS compatible to VDX) is not in contradiction to serializability. Parallel derivations (VD compatible to VD) are also secure, provided the checkin actions do correctly insert the corresponding derivation relationships[2]. Here, we use a special variant of multi-level transactions [28] providing serializability. We just need to consider two levels, the upper represented by the DOP and the lower represented by a transaction performing the checkin operation. Thus, parallel insertions of successors of the same input version are correctly synchronized by the lower level exclusively locking the version for short duration in order to insert the derivation information. These locks are just needed during the checkin phase and can be released at the end of the corresponding subtransac-

1. Such information can be computed by considering statistical information about previous runs of the considered tool.

2. We are modeling derivation relationships by special reference attributes (prior and next).

tion. Manipulating a version means at least being allowed to access the corresponding object in shared mode. Since each access to a version must first acquire an object level lock, this is also in accordance with serializability.

5. Conclusions

In this paper, we introduced the C^3-locking-protocol being the concurrency control mechanism used within the CONCORD activity framework. It contributes to enforce the ACID principles for long transactions manipulating versions. In our opinion, the protocol is adequately adapted to the special requirements of the surrounding processing structures and the underlying version-management layer:

• Version management: The transactions to be synchronized have a long system residence time and are manipulating the data modeled within the OVDM. Major principles of this model are that versions are apparent and that users view objects as sets of explicit versions. The corresponding query language OQL exploits the notion of complex structures (network-like). For these reasons, an adapted granular lock protocol would not be appropriate.

• Design-flow management / design-tool execution: design-flow management can be distinguished from workflow management, because design flows are typically less strict and less regular than workflows. Therefore, there must be interaction between design-flow management system and designer at run time, enabling the designer to bring in his skills, also in questions of flow of processing.

Besides that, design tools are usually highly interactive. This principle of interactive work can also be used to avoid unacceptable waiting situations. First, a communication between lock manager and tool is supported allowing to flexibly react to the case that (parts of) the wanted data is already locked in incompatible mode. In the case that those reactions cannot be foreseen (and embedded into the transaction program) there is still the possibility to extend the mentioned communication to the designer. We think, we do not burden the designer with those questions unnecessarily, because the modes are very intuitive and the designer is completely aware of the design-flow structures and the cooperation relationships to other DAs. He can best judge alternative ways of processing.

An additional requirement, our model introduces for the tool programmer, is the necessity of prespecifying the wanted access mode within checkout statements. Since in engineering application mostly processing contexts of tools are known in advance, the access mode can also be pre-determined.

• Cooperation control: Cooperation actions affect the concurrency control mechanism in the way that the locking protocol has to additionally check, whether the considered transaction is currently (w.r.t cooperation specifications) allowed to access the requested data in the requested mode. Therefore, it has to be aware of access rights granted along cooperation relationships. The enforcement of application-specific correctness criteria (implicit cooperation) does not touch concurrency control directly; this can be done exclusively at the cooperation layer.

Facing these side conditions, we think that the C^3-protocol is well suited for a structured activity model as the CONCORD model providing version management, transaction processing, design-flow management and cooperation control at separate system layers.

Literature

[1] Agrawal, D.; Bruno, J.L.; El Abbadi, A.; Krishnaswamy, V.: Relative Serializability: An Approach for Relaxing the Atomicity of Transactions; Proc. ACM SIGACT/SIGMOD Symposium on Principles of Database Systems, 1994.

[2] Bernstein, P.A.: Repositories Internals, Tutorial Handouts, 21st Int. VLDB Conf., Zurich, 1995.

[3] Bernstein, P.A., Hadzilacos, V., Goodman, N.: Concurrency Control and Recovery in Database Systems, Addison-Wesley Publishing Company, MA, USA, 1987.

[4] Barghouti, N.S., Kaiser, G.E.: Concurrency Control in Advanced Database Applications, ACM Computing Surveys, Vol. 23, No. 3, September 1991.

[5] Breitbart, Y., Deacon, A., Schek. H.J., Sheth, A., Weikum, G.: Merging application-centric and data-centric approaches to support transaction oriented

multi-system workflows, SIGMOD RECORD, 22(3), 1993.

[6] Brettschneider, F.: A Process Model for Design Flow Management and Planning, Fortschritt-Berichte VDI, Reihe 9; Elektronik, Nr, 157, VDI-Verlag, Düsseldorf, 1993.

[7] Ellis, C.A.; Gibbs, S.J.; Rein, G.L.: Design and use of a group editor; Engineering for Human Computer Interaction; Cockton, G., Ed., North Holland, Amsterdam, 1990, pp 13-25.

[8] Elmagarmid, A. (ed.): Transaction Models for Advanced Database Applications, Morgan Kaufmann, San Mateo, CA, 1992.

[9] Georgakopoulos, D., Hornick, M., Sheth, A.: An Overview of Workflow Management From Process Modeling to Infrastructure for Automation, Journal on Distributed and Parallel Database Systems, 3(2), April 1995.

[10] Gray, J., Reuter, A.: Transaction Processing: Concepts and Techniques, Morgan Kaufmann Publ., San Mateo, CA, 1993.

[11] Grudin, J.: Computer-Supported Cooperative Work: History and Focus, IEEE Computer, May 1994, Vol. 27, No. 5, pp. 19-26.

[12] Harrison, D., Newton, R., Spickelmier, R., Barnes, T.: Electronic CAD Frameworks, Proc. of the IEEE, Vol. 78, No. 2, Febr. 1990, pp. 393-417.

[13] Härder, T., Mitschang, B., Ritter, N.: An Infrastructure for Cooperative Work Arrangements, Proc. Int. Workshop on Concurrent/Simultaneous Engineering Frameworks and Applications, Lisboa, 1995, pp. 407-424.

[14] Härder, T., Reuter, A.: Principles of Transaction-Oriented Database Recovery, ACM Computing Surveys 15, 4, 1983, pp. 287-318.

[15] Katz, R.: Toward a Unified Framework for Version Modeling in Engineering Databases, ACM Computing Surveys, Vol. 22, No. 4, Dezember 1990, S. 375-408.

[16] Kung, H., Robinson, J.: On optimistic methods for concurrency control, ACM Trans. Database Syst., 6, 2 (June), 1981, pp.213-226.

[17] Käfer, W., Schöning, H.: Mapping a Version Model to a Complex Object Data Model, Proc. 8th Int. IEEE Data Engineering Conference, Tempe, Arizona, 1992.

[18] Krishnakumar, N., Sheth, A.: Specification of Workflows with Heterogeneous Tasks in METEOR, Poster Paper Collection, 20th Int. VLDB Conf, Santiago, Chile, 1994.

[19] Leymann, F.: Supporting Business Transactions via Prtial Backward Recovery in Workflow Management Systems, G. Lausen (Hrsg.) Datenbanksysteme in Büro, Technik und Wissenschaft, Springer, 1995.

[20] Nink, U, Ritter, N.: VSTORE - Implementing a Version Management System on top of ObjectStore, Internal Report, University of Kaiserslautern, Germany, 1996.

[21] Nodine, M.H.; Zdonik, B.: Cooperative Transaction Hierarchies: Transaction Support for Design Applications; VLDB Journal 1, 1992, pp. 41-80.

[22] Ritter, N., Mitschang, B., Härder, T., Gesmann, M., Schöning, H.: Capturing Design Dynamics - The CONCORD Approach; Proc. 10th Int. IEEE Data Engineering Conference, Houston, Texas, 1994, pp. 440-451.

[23] Ritter, N., Mitschang, B., Härder, T., Nink, U.: Unterstützung der Ablaufsteuerung in Entwurfsumgebungen durch Versionierung und Konfigurierung, in Proc. STAK'94, Softwaretechnik in Automatisierung und Kommunikation - Datenbanken unter Realzeit- und technischen Entwicklungsanforderungen, 1994, Ilmenau, pp. 135-169.

[24] Reinert, J., Ritter, N.: Applying ECA-Rules in DB-based Design Environments, Internal Report, University of Kaiserslautern, Germany, 1995.

[25] Sutton, P.R., Brockman, J.B., Director, S.W.: Design management using dynamically defined flows, Proc. 30th ACM/IEEE Design Automation Conference, 1993, pp. 648-653.

[26] Sheck, H.J., Sheth, A.P., Czejdo, D.(eds): Proceedings RIDE-IMS '93, 3rd Int. Workshop on Research Issues in Data Engineering: Interoperability in Multidatabase Systems, Vienna, Austria, 1993.

[27] Wächter, H.; Reuter, A.: The ConTract Model, in: Transaction Models for Advanced Database Applications, Morgan Kaufmann, San Mateo, CA, 1992.

[28] Weikum, G.; Schek, H.J.: Concepts and Applications of Multilevel Transactions and Open Nested Transactions, in: Transaction Models for Advanced Database Applications, Morgan Kaufmann, San Mateo, CA, 1992.

[29] Zimmermann, G.: PLAYOUT - A Hierarchical Layout System, Proc. of the 18th GI Conference, Hamburg, Springer-Verlag, Informatik-Fachberichte 188, 1988, pp. 31-51.