1

Transaction-Based Design Data Processing in the PRIMA Framework

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

Abstract

Nowadays database technology is a key concept of integrated design environments and CAD frameworks. Crucial aspects of the exploitation of database systems in this application area are the modeling of design data as well as the modeling of activities managing this data. The latter point requires the development of transactional structures reflecting the actual processing characteristics of the various classes of design data in an adequate manner and, additionally, some support for controlled cooperation among designers. In this paper, we will discuss the distinct processing characteristics of the most important types of design data and will propose adjusted transactional structures reflecting the requirements of their manipulation. Furthermore, we will detail how these transactional structures can be embedded into a comprehensive processing model for design processes, which also supports cooperation control and design flow management besides data processing performed by a set of design tools.

1 Introduction

The term *CAD framework* [HNSB90, RS92] comprises all facilities supporting the CAD tool developer, the CAD system integrator as well as the system user (designer). Hence, the aspects of design data representation and design dynamics support are crucial issues. Design data representation calls for an adequate management of multiple types of data which can be classified by structural criteria (flat tuples vs. complex structured data) as well as application-specific criteria (versioned data vs. version-free data) [Ka90, WGLL92]. Capturing design dynamics requires a powerful processing model, which on one hand supports cooperation between designers and on the other hand keeps the design consistent. To accomplish the various aspects of data consistency, the approved transaction processing concept of database systems (Atomicity, Consistency, Isolation, Durability) [HR83] can be applied; it guarantees atomicity and serializability for a transaction which, for this purpose, isolates its execution from other transactions. For this reason, cooperation - a fundamental prerequisite in CAD frameworks - is not a natural property of ACID transactions. However, it may be supported by introducing a hierarchically structured processing model where ACID transactions are essential building blocks of the control structure. To solve cooperation control and data consistency, we propose to enhance the approved transaction concept by explicit cooperation control facilities and by further mechanisms supporting design management and design flow management.

PRIMA Framework

PRIMA Framework is a CAD framework which uses the structurally object-oriented database system PRIMA [Sch93] as integrated design data repository. According to [HNSB90] a CAD framework offers a number of services which are supported by the PRIMA framework as subsequently outlined[1]. On top of PRIMA, we implemented an object and version management system [KS92] providing means for the manipulation of explicit complex-object versions as well as for the manipulation of configurations (*version and configuration services*). The object and version management system offers the Object Query

[1] We will not consider the most elementary framework services of process management and physical data management constituting a common interface between the operating system and the framework itself.

F. J. Rammig et al. (eds.), *Electronic Design Automation Frameworks*

Language (OQL) at its interface which is embedded into a higher programming language. An adequate processing concept providing means for workstation/server communication and processing of cached design data on the workstation side facilitates the direct implementation of design tools as well as the implementation of foreign tool interfaces (*tool integration services*). During the design process, the design tools can be applied in a controlled manner. To pre-plan and schedule design tool applications *design-flow management services* are provided. The intention of these services is to guide users through complex activities and to apply certain design methodologies. Design-flow specifications can be (re-)used in multiple design tasks or design processes (*design methodology services*). Further services, e.g. *user-interface services*, are currently elaborated, but are beyond the scope of this paper.

In this paper, we focus on information sharing in cooperative work arrangements and the corresponding activity control aspect. Thus, we will first outline the major concepts of the object and version data model (Sect. 2). Afterwards an overview of the CONCORD processing model will be given (Sect. 3) capturing design dynamics and, therefore, comprising data management services (transactional concepts), design-flow management services as well as design-management and cooperation-control facilities. In Sect. 4, we will discuss transactional control structures allowing for the processing of design data. Regarding the processing characteristics of the data we will select the most adequate transactional structure and we will describe how this structure is embedded into the comprehensive CONCORD model. The last section (Sect. 5) gives a conclusion.

2 Object and Version Data Model (OVDM)

In [KS92], the object and version data model (OVDM) is proposed for the management of design data. We will only be able to 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 entirety, 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 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. Remember 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 integrity constraints are 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 [Ka90]. *Configurations* (not shown in Fig. 1) are occurrences of spec-

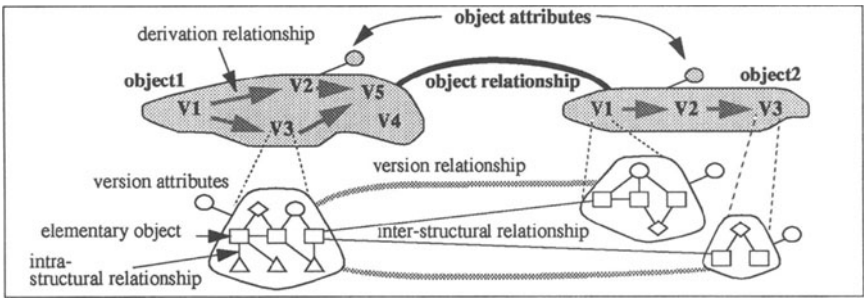

Figure 1: Major Concepts of the OVDM

ified configuration types which are usually associated with special integrity constraints expressing the requirements of the unit of versions to be established during the configuring activity. 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 embody 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 every 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.

The OVDM concepts, outlined in this section, are used in the PRIMA framework to model design data. In the following section, we will give an overview of the activity model used to process these data structures.

3 CONCORD Processing Model

The CONCORD[2] model [Ri94a] captures the dynamics inherent to design processes. To reflect the spectrum of requirements, such as *hierarchical refinement, goal orientation, stepwise improvement and team orientation*, three different levels of abstraction are distinguished as roughly illustrated in Fig. 2.

Administration/Cooperation Level (AC Level)

At the highest level of abstraction, we reflect the more creative and administrative part of design work. There, the focus is on the description and delegation of design tasks as well as on a controlled cooperation among the design tasks. The key concept at this level is the *design activity (DA)*. A DA is the operational unit representing a particular design task or subtask. During the design process, a *DA hierarchy* can be dynamically constructed resembling (a hierarchy of) concurrently active tasks. All relationships between DAs essential for cooperation are explicitly modelled, thus capturing task-splitting (cooperation relationship: *delegation*), exchange of design data (cooperation relationship: *usage*), and negotiation of design goals (cooperation relationship: *negotiation*). The inherent integrity constraints and semantics of these cooperation relationships are enforced by a central system component, called *cooperation manager*.

Design-Control Level (DC Level)

Looking inside a DA reveals the DC level. There, the organization of the particular actions to be performed in order to fulfill a certain (partial) design task is the subject of consideration (*design flow*). At this level, Fig. 2 shows an execution plan (*script*) of a particular design activity. The corresponding

[2]The CONCORD acronym stands for: CONtrolling COopeRation in Design environments.

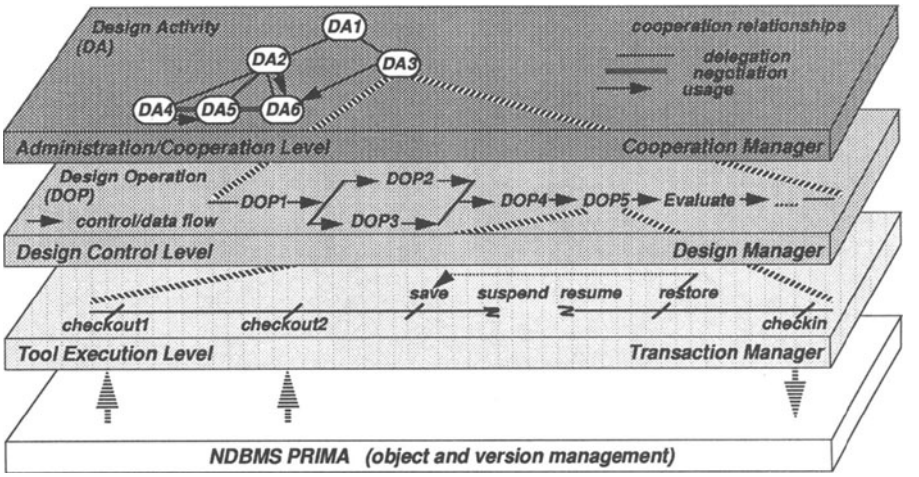

Figure 2: Abstraction Levels of the CONCORD Model

script models the *control/data* flow among several design actions performed within a DA. Usually, these actions are design tool applications. The operational unit serving for the execution of a design tool is the *design operation (DOP)*. In order to control the actions within the scope of a single DA, but without restricting the designers' creativity, flexible mechanisms for specifying the design flow for a DA (scripts, constraints, event-condition-action rules) are provided. The correctness of tool executions is guaranteed by a system component, called *design manager*. The design manager does also provide for recoverable script executions that is needed for level-specific and isolated failure handling. Design tools are applied to improve existing design states in order to finally reach at a design state that completes the current (partial) design task. Design states are captured by means of the object and version model introduced in the previous section. The derivation of versions and corresponding design object states by means of tool applications is supported by the concepts provided at the TE level.

Tool-Execution Level (TE Level)

From the viewpoint of the DBMS or data repository, a DOP is a long transaction (see in Fig. 2). A DOP has the properties of conventional transactions. Because of long duration, it is internally structured by save/restore and suspend/resume facilities [HHMM88] to be able to rollback the design at the application level and to continue the design work after breaks. A DOP processes design object versions in three steps. First, the input versions are checked out from the integrated data repository and cached in an object buffer at the workstation for efficiency reasons. Second, the design data is mapped to storage structures tailored to the application needs. It is processed by one or more design tools. Third, the finally derived new versions are propagated back to the data repository (check-in operation). The derivation of schema-consistent and persistent design object versions is guaranteed, again, by a central system component, called *transaction manager*. It is also responsible for the isolated execution of DOPs and for recoverable DOP executions that are, again, necessary for a level-specific and isolated failure handling. The transaction manager employs mechanisms provided by the advanced DBMS which manages the integrated data repository. Since long transactions are involved, incompatible lock requests on data already locked for an anticipated long duration[3] cannot be handled by usual wait mechanisms, but require a notification concept.

[3]In the case of version derivations, generally long locks are not required, because multiple successors of a version can be derived concurrently.

Overall Architecture

At the conceptual level, all information relevant for a design process is managed by one logical server so that no exchange of design data between distributed databases becomes necessary during the design process and a cooperative data exchange between concurrently active design tasks can be managed via the integrated data repository. The logical server, in turn, may be implemented by a set of distributed and heterogeneous physical servers; this mapping has to be achieved by the DBMS which encapsulates the interoperability problems from the logical server view of design data management. In contrast, the designer carries out his design work at a (single- or multi-processor) workstation. Since a DA typically comprises the design work of a single designer, we assume that a DA is running at a single workstation. Consequently, all actions executed within a DA are managed and executed at this workstation, too. This is needed for three reasons. Firstly, in many cases the designer has to specify input parameters for the design tools. Secondly, designer interaction during tool execution is necessary and essential, and, thirdly, the information derived by a DOP is mostly subject to work-flow (data-flow) management within the DA.

So far, we have outlined data modeling and activity modeling aspects as well as some architectural issues. In the following section, we will examine the needs of the application pertaining the modeling and the management of various classes of design data in more detail. This will lead to a refinement of the transactional control structures used in the CONCORD model.

4 Transaction-Based Design Data Processing

In this section, we want to discuss transaction-based processing of design data in a little more detail. CONCORD provides a very coarse view to design data processing. The processing model must be refined regarding the various types of design data accessed during the design process. Therefore, we first want to classify the most important types of design data. Afterwards, we will detail the CONCORD processing model.

First of all, however, we have to redefine the notions of transaction commitment and data release, because they are used in slightly different meaning. Committing a DOP means making schema-consistent data persistent, but it does not imply that this data becomes automatically visible to any user. It remains associated to the corresponding DA. Exchanging data between DAs or making data visible to other DAs is captured by the notion of data release. Here, two cases can be distinguished. First, with the successful termination of a DA, its results (final design data of the DA) are passed to the parent DA. Second, preliminary design data can be exchanged between DAs via the usage relationship enforcing a special cooperation protocol. Final design data of the overall design process becomes generally visible with the successful termination of the top-level DA.

Classes of Design Data

During a design process two major classes of design data are accessed. We call the first class *task-related data (DA-related data)*. This data can be characterized as follows:

- it is subject of design work (stepwise improvement); it represents the design objects of DAs and captures the different design object states developed during the design;
- it is usually complex structured, versioned and manipulated by long transactions exploiting check-out/ check-in primitives; it is useful to manage different states of task-related data in order to be able to return to an older design state as a starting point for the development of a better solution;
- it is subject to various kinds of cooperation; first, there is a restricted release of the final design data of a DA to the parent DA (along the delegation relationship); second, preliminary design data can be exchanged among DAs (via usage relationships); third, design objects can be shared among certain DAs (*object pools*); they can manipulate them in a mutual manner;
- the design process is terminated by an unrestricted release (successful termination of the top-level DA); if the data has to be further protected, access control mechanisms seem to be appropriate.

The second class (*common design data*) comprises the data which is not related to a single design process. For example, information libraries, general rules, regulations, and catalogs about standards belong to this class. Furthermore, information about technology, tolerances, materials, etc., is contained in this class. This data is typically accessed in the following way[4]:

- read access is predominating; modifications are infrequent;
- information requests can usually be handled by means of short transactions;
- there can be concurrent and independent requests of a single designer in parallel to his design tool applications.

However, upon arrival of new rules, standards, etc., or when errors are detected, the common design data is occasionally updated by the database administrator or even by the design transactions. These changes have to be performed atomically.

Before detailing the CONCORD processing model in order to be able to adequately process both classes of design data, we want to outline how CONCORD exploits OVDM concepts to manage task-related data.

Interplay of CONCORD and OVDM

The discussion of the characteristics of the task-related data indicates that CONCORD and OVDM are closely interplaying. The design objects manipulated by the DAs of a DA hierarchy (or their DOPs, respectively) can adequately be modelled by means of OVDM concepts.

At creation time, every DA is associated with a *design object type*. The design object type is usually given as a set of complex-object types which are connected by object relationship types forming a configuration type. Thus, the objects and versions manipulated by a certain DA in order to reach its design goal are occurrences of the complex-object types contained in its design object type. This data is processed within DOPs. A DOP receives as input a set of versions, does some computations, and produces as output a schema-consistent set of versions (see TE level). We are considering this set of versions as an implicit (and mostly partial) *design object state* of the corresponding DA. An explicit (and complete) design object state is given as an occurrence of the configuration type constituting the design object type of the DA[5]. The creation of preliminary and final design object states as well as the synthesis of partial results delivered by sub-DAs can adequately be done by means of the mentioned configuration concept. Usually, the hierarchical structure of configuration types (sub-configurations) is a natural basis for the decomposition of design tasks associated with DAs.

Detailing the Processing Model

In distributed environments (such as workstation/server architectures), there exist two fundamental principles to process data: either *the functions follow the data* or *the data has to be moved to the functions*. The processing characteristics of the data determines, which principle has to be selected to achieve optimal performance. For this reason, adjusted processing models have to observe the predominating properties of data access.

Considering the processing structures in a little more detail, we have to incorporate control structures for accessing task-related data as well as common design data. From the viewpoint of the DAs, common design data is typically subject to information requests. Despite this predominating read access we also have to consider updates on common design data initiated by DAs. In every case, however, it is not necessary to cache and process common design data at the workstation, because locality of references cannot be anticipated in contrast to the processing of task-related data. Thus, we have the following aspect distinguishing the processing of the two mentioned classes of design data. The manipulation of

[4] Archived design data (from earlier design processes) has similar processing characteristics.

[5] Since the creation of configurations is expected to be done in the later phases of the design process, we are also considering the results of DOPs as preliminary design object states which in turn can be subject of cooperation between DAs.

task-related data can adequately be supported by caching it at the workstation and providing an appropriate processing concept for navigational and repetitive access (e.g., hierarchical cursors [HS92]). The manipulation of common design data, in contrast, is best performed by sending a function call to the server, which then computes the function using the stored data and sends back the results to the corresponding workstation.

So we have to raise the question on how to adequately organize a DA's access to the various types of design data. Regarding the needs of the application, we first of all have to consider, whether it is sufficient to provide special and independent application programs allowing the designer to do information requests (separate processes for information requests and DOPs[6]) or whether it is necessary to incorporate these information requests into the tools processing the task-related design data (a single process). In any case, appropriate transactional structures have to be supported [SI93].

In the following, we will examine different transactional structures in order to find an adequate one for the *combination of DOP processing and information requests.*

A single long transaction. Handling information requests within the long transaction processing the taskrelated data leads to a very simple transactional structure. The drawbacks are obvious. The locks on the common design data must be held until commit of the long transaction in order to provide correct (serializable) schedules [BSW79]. This could be an unnecessary long period during which blocking of concurrent (long) transactions is not acceptable and which would require frequent notifications even for accesses to common design data.

A single nested transaction. The nested transaction model [Mo81] allows the creation of a transaction tree. The subtransactions are atomic and are isolated against each other. Thus, the model allows for the parallel execution of the actions performed within sibling transactions and for a fine granular recovery since subtransactions can be aborted without any effect on the parent or sibling transactions. Only the top-level transaction has all ACID properties, because all locks hold by a committing subtransaction are upward inherited. This model would allow to perform actions processing task-related data and actions on common design data in separate subtransactions so that the abort of one transaction would not affect the other actions. Despite this higher flexibility we still have the drawback that all locks, and in particular the locks on common design data, are to be held till commit of the top-level transaction.

A single nested transaction with open subtransactions. An extension of the nested transaction model (inspired by the sagas model [GS87]) proposes to distinguish closed subtransactions from open subtransactions. Closed subtransactions are those of the original model passing the locks to the parent transaction at commit. Open subtransactions, on the other side, release their locks at commit (of the subtransaction) so that the corresponding data becomes accessible for concurrent (nested) transactions. This model allows for handling access to common design data within open subtransactions so that the corresponding common design data can be released at commit of the subtransaction. The drawback of this model is that it requires a compensation transaction for every open subtransaction allowing for a semantic UNDO of the subtransaction after its commit. This becomes necessary, whenever a higher level transaction aborts. This abort dependency would be only adequate, if there would be causal dependencies from the information request to the manipulation of task-related design data. We don't see these causal dependencies. For that reason, it is not necessary to provide compensation transactions so that this model is not adequate, too.

A single activity/transaction hierarchy. The ATM (activity/transaction model [DHL91]) provides very flexible mechanisms for the creation of activity/transaction hierarchies. This model enhances the mentioned nested transaction model. Additional operational units called ***activities*** can be incorporated which do not have transactional properties. For example, accesses to read-only data (bulletin board, etc.) could be performed in activities as well as operations not related to database data. Activities and transactions can be arbitrarily nested except nesting activities into transactions. In this way, pure nested transactions are establishing subhierarchies of an activity/transaction hierarchy. The relationship

[6] We use the term "DOP" synonym to "manipulation of task-related data" and "information request" synonym to "manipulation of common design data".

resulting from nesting a subtransaction into an activity is characterized as follows: The transaction has access to all objects manipulated by the activity and the activity can only terminate after the termination of the transaction. A further feature of this model is the possibility to define causal dependencies between activities/transactions. If unit A is causally dependent from unit B this implies that ABORT(B) will lead to ABORT(A). There are two ways to model the manipulation of design data by the concepts of the ATM model. The first way is to treat both, the manipulation of task-related data as well as the manipulation of common design data, as (sub)transactions. As already mentioned, there are no causal dependencies from the manipulation of common design data to the manipulation of task-related data, so that this feature cannot be exploited and the model would degenerate to the pure nested transaction model which is not adequate for the reasons mentioned above. The second possibility is to treat access to common design data as activities and not as transactions. This is also not appropriate because the manipulation of common design data should also be protected by transactional mechanisms.

Multiple independent transactions. The last transactional structure for combining the manipulation of taskrelated data and the manipulation of common design data is a very easy one providing high flexibility. While the access to common data is performed within a short ACID transaction the processing of task-related data is encapsulated by an independent long transaction. This simple structuring ensures that the different processing characteristics do not influence each other negatively. For that reason, we have chosen this last possibility. It enforces to mark the transaction types already during the implementation of the application programs and design tools because the distinct types of transactions are implying distinct processing models. While the long transaction, which is well suited for the processing of task-related data, usually complex structured and versioned, exploits check-out/check-in primitives as well as caching and processing of design data at the workstation side (see properties of DOPs), the conventional short transaction, which is well suited for the manipulation of common design data, can be seen as a function call which is executed at the server and causes only the results to be transferred back to the workstation. In order to support adequate supply of design tool applications (encapsulated by long transactions) with context information as well as information derived from common design data, adequate parametrization of design tools and adequate data-flow capabilities must be provided.

Data Flow

The intuitive model of data flow in CONCORD is that long transactions propagate the newly derived versions to the data repository by check-in operations so that data supply of succeeding transactions can be managed via check-out operations. This model does not provide good performance in the case where the data produced by a certain tool is needed as input for a directly succeeding tool. Therefore, we propose the following optimization. The new versions produced by a DOP, which are needed by a succeeding DOP, remain in the workstation cache, so that they can be further processed by the succeeding DOP. In order to provide durability of these versions we have to write a persistent log at the workstation side. The server just gets the information that new versions are derived and generates the according version numbers and housekeeping information. By this mechanism, it is possible to asynchronously transfer the new versions to the server (e.g., at the latest, when they are accessed by a cooperating DA performed at another workstation). The mentioned actions for check-in of a DOP are appropriate, because the results of long transactions remain task-related, although they are to be stored persistently in the data repository. The results remain associated to the corresponding DA and can only be made visible to other DAs via cooperation primitives.

Concurrency/Cooperation-Control

As mentioned previously all (short and long) transactions, independently to which DA they belong, are executed in isolation. Short transactions are manipulating common design data at the server side and their results are released at commit. Long transactions (DOPs), on the other hand, are processing their data at the workstation side, their results remain task-related and the data release depends on the cooperation relationships of the corresponding DA.

Regarding versioning, synchronizing long transactions becomes quite easy. As compared to the usual readwrite synchronization on version-free data, versions allow for much higher concurrency as illustrated by the lock compatibility matrix in Fig. 3. Generally, the concurrent derivation of several successors of the same input version does not lead to any conflicts. Despite of that, it may be necessary in special

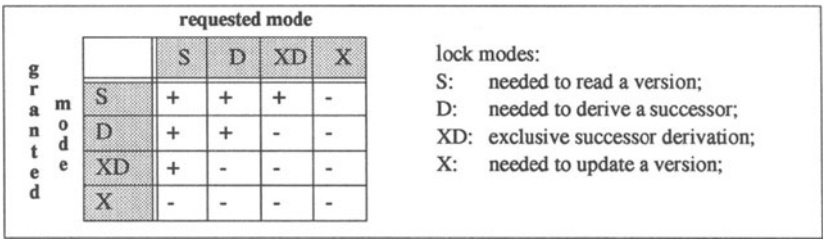

Figure 3: Lock Compatibility Matrix for Version Manipulations

cases to prevent parallel derivations from the same input version for application-specific reasons. Here, as well as in the case of version updates (which occur very seldom) blocking is required. In order to avoid an unreasonably long waiting situation, which can be anticipated whenever a check-out operation cannot acquire the required read or derivation locks, this situation is reported to the tool. This allows the tool implementor to foresee appropriate contingency actions, e.g., notification of the designer.

As already mentioned at the beginning of this section, the release of task-related data depends on the cooperation relationships of the corresponding DA hierarchy. Especially along usage relationships the creating DA may permit cooperating DAs to access propagated design data. The kind of access to cooperatively exchanged data depends on the chosen protocol. For example, it could be sufficient to only permit read access; in other cases, it could be appropriate to let the cooperating DA derive new versions and to give them back as some kind of proposal. Since the discussion of these protocols is beyond the scope of this paper (see [Ri94a, Ri94b]), we just want to mention that the server has to manage the cooperation relationships between the DAs. Thus, the server is aware of the release state of task-related data and can decide whether a certain DA's DOP can be permitted to access a requested version.

5 Conclusions

In this paper, we have been focusing on the data modeling and data manipulation aspects in cooperative design applications. The CONCORD processing model which allows the straightforward mapping of the processing structures predominating in design processes and, therefore, serves as the processing model of the PRIMA framework, has been refined so that processing of the most important types of design data can adequately be encapsulated within ACID transactions. The concepts of the object and version data model OVDM are well suited to model these classes of design data. Thus, CONCORD and OVDM are closely interplaying in order to meet the requirements of the application area.

Furthermore, CONCORD incorporates cooperation control and design-flow capabilities. To support collaboration explicit control mechanisms supporting a user-centered cooperation [Ri94b] are combined with traditional concurrency control mechanisms. This, on one hand, allows a very flexible cooperation among designers, and, on the other hand, keeps the design data consistent, because approved transaction concepts are applied. The design-flow features could not be detailed in this paper. This matter concerns pre-planning and scheduling complex activities in order to guide the user and help him to concentrate on the creative design decisions he has to take. We are currently working on this topic in order to find appropriate control structures to connect the transactional steps processing design data (cf. [WR92], [RW92]).

The aspects mentioned in this paper address major objectives of the DoCAIA project (Brazilian/German Cooperation on the "Support of Design and Development of Computer Applications in Industrial Automation"), so that the results discussed in this work can be a foundation of a further cooperation in the context of this project.

References

[BSW79] Bernstein, P.A.; Shipman, D.W.; Wong, W.S.: Formal Aspects of Serializability in Database Concurrency Control, IEEE Transactions on Software Engineering, Vol 5, No. 5, May 1979, pp. 203-216.

[DHL91] Dayal, U., Hsu, M., Ladin, R.: A Transactional Model for Long-Running Transactions, Proc. 17th Int. Conf. on VLDB, Barcelona, Spain, 1991, pp. 113-122.

[GS87] Garcia-Molina, H.; Salem, K: Sagas, Proc. ACM SIGMOD, 1987, pp. 249-259.

[HHMM88] Härder, T., Hübel, C., Meyer-Wegener, K., Mitschang, B.: Processing and Transaction Concepts for Cooperation of Engineering Workstations and a Database Server, Data and Knowledge Engineering 3, 1988, pp. 87-107.

[HNSB90] Harrison, D., Newton, R., Spickelmier, R., Barnes, T.: Electronic CAD Frameworks, Proc. of the IEEE, Vol. 78, No. 2, Febr. 1990, pp. 393-417.

[HR83] Härder, T., Reuter, A.: Principles of Transaction-Oriented Database Recovery, ACM Computing Surveys 15, 4, 1983, pp. 287-318.

[HS92] Hübel, C., Sutter, B.: Supporting Engineering Applications by New Data Base Processing Concepts - An Experience Report, Engineering with Computers, 8, pp. 31-49, 1992.

[Ka90] Katz, R.: Toward a Unified Framework for Version Modeling in Engineering Databases, ACM Computing Surveys, Vol. 22, No. 4, Dezember 1990, S. 375-408.

[KS92] Käfer, W., Schöning, H.: Mapping a Version Model to a Complex Object Data Model, Proc. 8th Int. Conf. on Data Engineering, Tempe, Arizona, 1992.

[Mo81] Moss, J.E.B.: Nested Transactions: An Approach To Reliable Computing, M.I.T. Report MIT-LCS-TR260, M.I.T., Laboratory of Computer Science, 1981.

[Ri94a] Ritter, N., Mitschang, B., Härder, T., Gesmann, M., Schöning, H.: Capturing Design Dynamics - The CONCORD Approach, Proc. 10th Int. Conf. on Data Engineering, Houston, Texas, Feb. 1994, pp. 440-451.

[Ri94b] Ritter, N.: An Infrastructure for Cooperative Applications based on Conventional Database Transactions, Proc. of the CSCW Infrastructure Workshop, Chapel Hill, North Carolina, Oct. 1994.

[RS92] Rammig, F. J., Steinmüller, B.: Frameworks and Design Environments, in: Informatik-Spektrum (1992) 15: 33-43 (in German).

[RW92] Reinwald, B., Wedekind, H.: Automation of Control and Data Flow in Distributed Application Systems, in: Tjoa, A., Ramos, I. (eds): Database and Expert Systems Applications - DEXA (Proc. of the Int. Conference, Valencia), Springer, Wien, 1992, pp. 475-481.

[Sch93] Schöning, H.: Query-Processing in Complex-Object Database Systems, Deutscher Universitäts-Verlag, 1993 (in German).

[SI93] Souto, M.A.M., Iochpe, C.: Transaction-Based Support for Production Plan Execution: a Position Paper, Proc. 1st Brazilian Symposium on Intelligent Automation, Aguas Claras, SP, Brazil, 1993.

[WGLL92] Wagner, F.R., Golendziner, L.G., Lacombe, J., Lima, A.V.: Design Version Management in the STAR Framework, in: Electronic Design Automation Frameworks, Newman, M. and Rhyne, T. (eds.), Elsevier Science Publ. (North-Holland), 1992, pp. 85-97.

[WR92] Wächter, H.; Reuter, A.: The ConTract Model, in: Transaction Models for Advanced Database Applications, Morgan Kaufmann, San Mateo, CA, 1992.