Data & Knowledge Engineering 3 (1988) 87–107
North-Holland

Processing and transaction concepts for cooperation of engineering workstations and a database server

T. HÄRDER, Ch. HÜBEL, K. MEYER-WEGENER and B. MITSCHANG
University Kaiserslautern, Erwin-Schrödinger-Straße, D-6750 Kaiserslautern, F.R. Germany

Abstract. A DBMS kernel architecture is proposed for improved DB support of engineering applications running on a cluster of workstations. Using such an approach, part of the DBMS code—an application-specific layer—is allocated close to the corresponding application on a workstation while the kernel code is executed on a central server. Empirical performance results from DB-based engineering applications are reported to justify the chosen DBMS architecture.

The paper focuses on design issues of the application layer including server coupling, processing model and application interface. Moreover, a transaction model for long-term database work in a coupled workstation-server environment is investigated in detail.

Keywords. Engineering databases, design transactions, workstation-server coupling, object processing, locality.

1. Introduction

Complex engineering tasks involve many related issues—of prime importance is the integrated management of design and product data describing all relevant aspects of the construction process as well as all essential properties (technological, physical, geometrical) of the design objects. Approaches which have combined engineering systems with traditional database management systems (DBMS) have suffered from a number of deficiencies. Major reasons of inappropriate DB support for all kinds of engineering applications are:

- Classical data models have only a limited capability for modeling complex objects; moreover, they provide insufficient operational support and integrity control.
- The transaction model is developed for typical data-processing applications. Each transaction as a unit of consistency is assumed to take a few seconds thereby immediately propagating all updates to the database.
- Engineering applications typically require interactive computing environments; nowadays, powerful workstations offer tailored support for such use. Straightforward coupling of engineering systems with DBMS running on a host (server) yields slow (remote) reactions when DB services are requested – for a variety of reasons. In particular, this kind of coupling is not adjusted to the local processing capabilities. Furthermore, it suffers from a severe lack of locality of data reference since local buffers are not exploited at all.

In this paper, we discuss an architecture for improved DB support tailored to engineering applications and for an integrated processing model coupling server and workstation clusters. To motivate this approach, at first we report on our performance experiences in various engineering areas which were gained by building sizable prototype applications on top of a conventional DBMS. Our empirical results suggest the use of the DBMS kernel architecture

[7, 15] where the top-most layer of the DBMS—called application layer—is allocated together with the application program to the specific workstation, while the kernel serving multiple applications layers is running on a server. Our main concern is the design of the application layer. We investigate the interface between kernel and application layer and develop models of object processing using local buffers, thereby preserving high degrees of locality. Furthermore, we propose a transaction model adjusted to the workstation—server environment and its failure situations.

2. Performance evaluation of DB-based engineering applications

Our overall goal is the investigation of suitable system architectures to connect engineering applications running on dedicated workstations with a DBMS allocated at a central server. To refine the operational requirements of such a coupling task, we studied the specific problems empirically using various prototype implementations. Therefore, we developed three different kinds of DB-based applications dealing with geometric objects:

- a 3D-CAD application for volume-oriented geometric modeling
- a VLSI design tool for supporting optimal chip planning
- a land information system managing geographic data.

Our prototype approach and the principal results gained are explained by referring to the CAD application. Fig. 1 illustrates the overall system architecture consisting of graphic I/O system, CAD application and data management component. For our purpose, we focus on the issues of data management. Since no appropriate engineering DBMS was available, we enhanced a conventional CODASYL DBMS by an 'additional layer' to obtain more powerful operations and data structures at the interface of the data management component (called *application-supporting interface*). Note that the 'additional layer' approach only

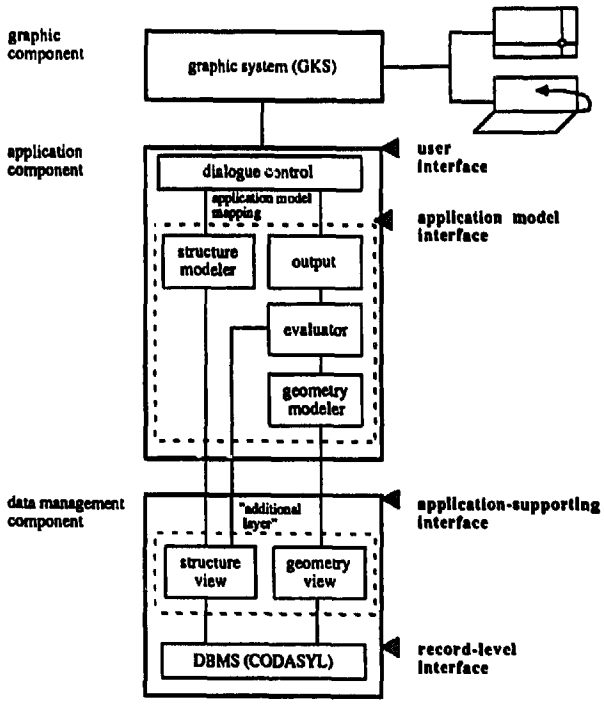


Fig. 1. Overall architecture of a CAD system for geometric modelling.

provides functions (tailored to a specific application) at its interface, but not the requried performance, since these functions are implemented on top of an unsuitable DBMS interface (called *record-level interface*).

The CAD application performs the construction of solids upon user requests from a graphical I/O interface. The chosen approach is based on a volume-oriented scheme called Constructive Solid Geometry (CSG [16]); the user is guided by menues, selects from predefined parametric base volumes and regular operators (union, difference, etc.) and, thus, composes his workpiece in consecutive steps.

Solids are represented in the DB by CSG trees which describe the corresponding history of construction. To facilitate graphical representation and special geometric operations, a dual representation called boundary representation (BREP) is automatically derived and maintained by the evaluator and stored in the DB. The required type information for the structure view and the geometry view is shown in Fig. 2 schematically as kind of Bachman-diagram for CODASYL data types—set names are dropped, relation records (mapping of n:m-relationships) are represented by small circles, cardinality restrictions indicate some structural integrity constraints. Note that Fig. 2 only illustrates the most important DB-schema part; a complete product model governing the construction process would include technological, physical and organizational submodels as well.

Let us refine our view of the geometry model which is used to indicate severe performance problems of geometric modeling based on traditional DBMS approaches. The kernel part of the geometric model is given by the record types BREP, Face, Edge, Point and the corresponding relationships (most of them are n:m). It allows for and guarantees object modeling without redundancy. However, the resulting representations may lead to tedious and poor modeling algorithms since implicit information, e.g. all edges belonging to a particular track, have to be derived over and over again. Hence, it may be advantageous to provide useful redundancy in order to simplify the modeling algorithms of the application component. On the other hand, such specific, deliberately designed redundancy increases the complexity of data management and enhances the mapping overhead thereby influencing the

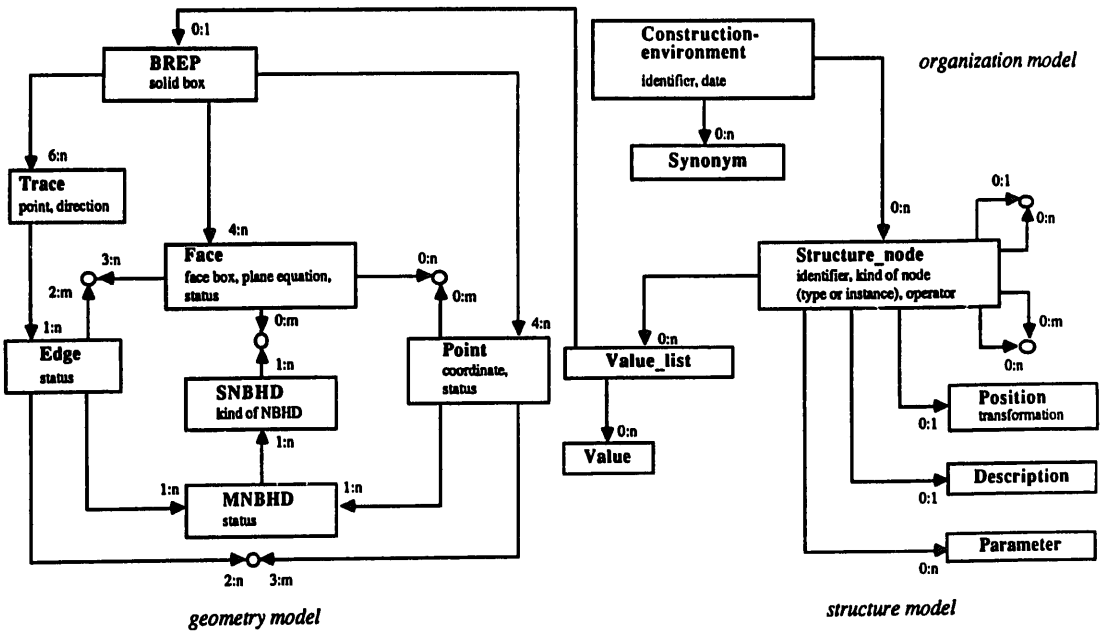
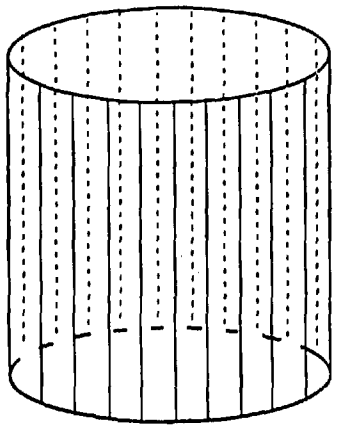


Fig. 2. DB schema of the CAD application.

performance of the data management component. With the hope of valuable insights, we therefore added the redundant record types Track collecting edges of particular faces, SNBHD and MNBHD (single and multiple neighbourhood) to describe the specific environments of points [5].

The view given by the DB schema is the one of the record-level interface (RLI). The corresponding manipulation language—the CODASYL DML—embodies navigational and record-oriented operations (e.g. FIND NEXT WITHIN SET) which frequently depend on cursor positions and set references. These interface properties should be kept in mind when the workload at the RLI created by geometric operations is evaluated and interpreted. As compared to the RLI, the gain of abstraction at the application-supporting interface (ASI) may be made clear by the example operation sketched in Fig. 3. The given program is executed at the ASI to generate a cylinder approximated by a polyeder. This program, in turn, is invoked at the application model interface (AMI) by a single operation POLYCYL(n) where n determines the number of lateral faces as an actual parameter.



```
Polycyl(n):

  ADDBREP ...          generation of a BREP-record
  ADDFACE ...          generation of the two Face-records, one for the top
  ADDFACE ...          face of the POLYCYL, the other for the bottom face
  Loop ADDFACE         generation of a Face-record for one side face
       Loop ADDEDGE    generation of one Track-, two Point-, and one
                       Edge-record as well as the generation of all
                       corresponding relation records
       End Loop        stop if all edges of one face are stored
       Loop CONFACE    construction of the neighbourhood information
                       for the common edges of adjacent faces
       End Loop        stop if the side face is "connected" to the top and
                       bottom face
       Loop SET_NBHD_OF_POINT  construction of the neighbourhood information
                               for the corner points
       End Loop        stop if all neighbourhood information is stored
  End Loop             stop if all faces of the POLYCYL are constructed
  UPBREP               actualization of the box attributes (solid box and
                       face box)
```

Fig. 3. Cylinder approximated by a polyeder and its generation procedure.

POLYCYL(n) is used as a first example to reveal the processing overhead needed to create the DB representation of cylinders—subject to various degrees of approximation. To simplify our investigations, we developed a fairly general measurement and evaluation tool which recorded events at different levels of abstraction. The results for POLYCYL(n) are summarized in Table 1. Hence, a cylinder generation with 50 lateral faces consumes 604 operations at the ASI and 41828 operations at the RLI.

Table 1. Call frequencies for the BREP generation of POLYCYL(n).

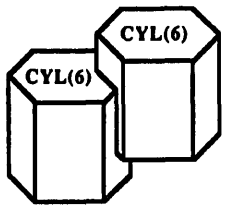
POLYCYL(n)	5	25	50
AMI	1	1	1
complete BREP-schema			
ASI	64	304	604
RLI: (a) simple subschema	1958	15078	41828
(b) use of locality	~1850	~10800	~25600
(c) use of context knowledge	~1450	~7400	~14800
Kernel part only			
ASI	39	179	354
RLI: (d) eliminated redundancy	1179	6159	11484

This vast amount of operations for such simple object creation is so incredible that it requires some further explanation and interpretation. The costs at the ASI is clearly linear with n, whereas the overhead at the RLI exhibits a strong non-linearity. Hence, ASI may be interpreted to represent a 'natural' interface for the required kind of application; of course, this is not true for RLI.

Table 1 additionally indicates some optimization efforts performed after thorough analysis of the measurement results.

a. The initial solution incorporates the subschema concept of CODASYL systems with at most one occurrence per record type under control and record-oriented manipulation. Reference typically leads to repeated DBMS calls due to lack of preserving locality in the working area above RLI.
b. The ASI-RLI mapping was modified to introduce a working area buffer to enhance locality of record reference, that is, more than one occurrence per record type was kept in the buffer if needed. Hence, many search operations within sets, etc. could be achieved without reaccessing the DBMS.
c. Another great reduction of overhead was realized by specializing ASI-operations by explicitly using context knowledge. Since this knowledge is application-dependent, it must be made available by the application.
d. In order to show the influence of our modeling redundancy, the operations related to the kernel part of the geometric model were explored separately. However, even this very simple and spartan modeling yielded more than 10^4 DBMS operations.

A second example is used to demonstrate the processing overhead during construction, that is, the costs for modifying and maintaining existing structures. Here, the generic operation is 'UNION (SOLID1, SOLID2)' where SOLID2 is derived from SOLID1 by a simple displacement. Fig. 4 sketches graphically the kind of operation. To facilitate comparison, the chosen solids always produce the same topological effects; the cost of the UNION operation (using the simple subschema concept) is listed for a few parameters (POLYCYL is abbreviated by CYL). Again, the performance figures indicate a tremendous mapping overhead.



Operation	#DMLs
UNION(CYL(3), CYL(3))	16051
UNION(CYL(4), CYL(4))	20658
UNION(CYL(6), CYL(6))	31432

Fig. 4. Union of two parameterized solids (Polycyl's).

To identify the inherent causes of the mapping problem, it was attempted to figure out the sources of overhead. As shown in Fig. 5, there exist some unique trouble spots. The n:m-relationships especially for Face and Edge as well as for Edge and Point are responsible for a huge portion of the overhead. This behaviour may be explained by successively inserting face-edge representations; in case of the front face n (50) edges have to be connected in various set occurrences, each one invoking positioning and store operations. The comparison of Fig. 5a and 5b reveals some important properties. Reference frequencies are not uniformly distributed; they vary depending on object types and kind of operation. For example, some relationships are not touched at all during object creation, wheras they are frequently used during object maintenance.

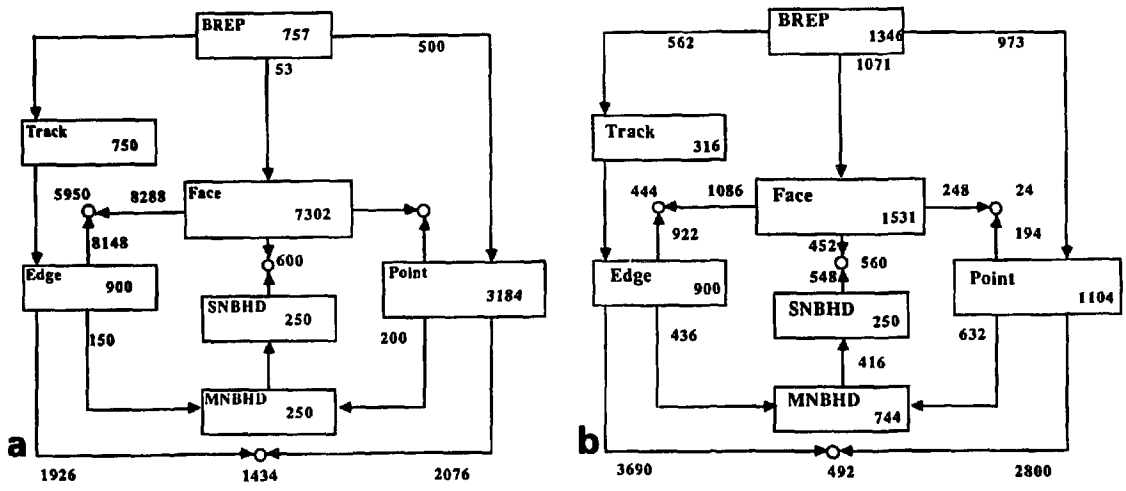


Fig. 5. Reference frequencies (#DMLs) for schema objects (non-optimized case):
(a) generation of POLYCYL (50)
(b) UNION (CYL (6), CYL (6)).

Neither processing optimizations nor reduction of modeling redundancy has confined the overhead to a reasonable limit, e.g. for interactive construction. Although a fraction of the costs is attributable to the properties of the CODASYL DBMS-interface—for example, frequent positioning of cursors, subschema concept and explicit access path navigation—the situation is not fundamentally improved by the other classical data models. Note, the relational model also requires a DB schema with a relation between two entity types associated by an n:m-relationship. Moreover, several thousand operations should be expected for POLYCYL(50) in the relational model—less than for the CODASYL model but more expensive ones, e.g. joins compared to FIND OWNER/FIND MEMBER.

3. Overview of the system architecture

In our other prototype applications, we have obtained similar performance results. To drastically reduce the overhead accompanying the work in a CAD environment, we were finally convinced by the derived performance figures that a better DB-approach to engineering applications should obey the following principles:

- The DB-interface to the CAD application should incorporate some object orientation, at least as powerful as the given application-supporting interface (Fig. 1).
- The data model interface should be substantially more powerful than the RLI; it should be set-oriented and should embody appropriate features for object handling support. Furthermore, tedious and cumbersome modeling of n:m-relationships should be avoided.
- Most important, locality of object processing should be preserved; it should support as close as possible the respective application.

A consequence of these requirements, observations, and ideas is the so-called DBMS kernel architecture [2, 7, 15], as illustrated in Fig. 6. Although at the first sight similar to the 'additional layer' architecture, closer consideration reveals a number of important differences. First of all, a strong separation is assumed between kernel and application layer. The kernel is defined to be application-independent. It realizes neutral, yet powerful mechanisms for supporting engineering applications which include storage techniques for a variety of object sizes, flexible representation and access techniques, basic integrity features, etc.

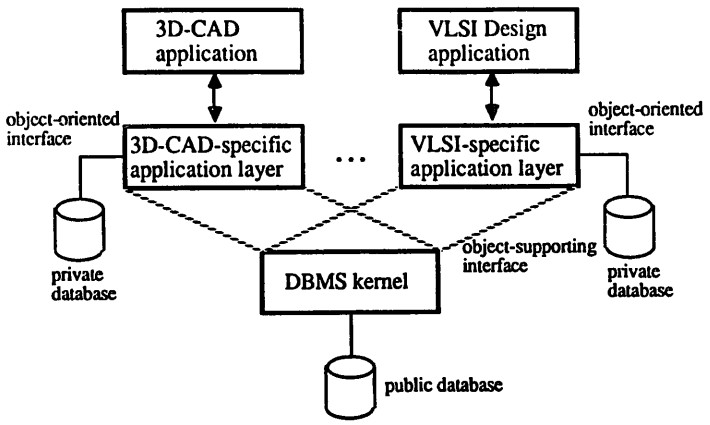


Fig. 6. DBMS kernel architecture—overall view.

The application layer (AL) achieves kind of tailoring mechanisms useful for specific applications. Since only neutral mechanisms are offered by the kernel, the required orientation towards the application implies that object orientation and most semantic issues, e.g. object-oriented representations and operations as well as integrity checks must be handled within the AL. Hence, the AL refers to lower level objects to create and manipulate application objects.

The clear division between kernel and AL is necessary for simultaneously providing a multiplicity of different application layers cooperating with the same kernel, as indicated in Fig. 6. Apparently, such a division simplifies the allocation of ALs to separate processors (e.g. workstations) and does not prohibit the integration of kernel and ALs in a single host environment.

For our purpose, it is only necessary to identify the various interfaces in slightly more detail. The data model interface is characterized as an object-supporting interface; it is assumed to incorporate the following properties:

- modeling techniques to specify the structure of a complex or composite object type consisting of various component types
- dynamic composition and decomposition of data structures belonging to different types (record types)
- set-oriented access to fetch or manipulate a set of heterogeneous records
- support for structural integrity checking.

The interface between engineering application and AL is qualified by the term 'object-oriented interface'. It is a higher level interface compared to the data model interface of the kernel. Its orientation is more towards application objects whereas the data model interface offers more or less neutral object support. Some essential properties of this interface are:

- application objects have an identity; they can be handled as integral entities
- such objects have an internal structure; reference to structured subcomponents is possible
- data abstraction and encapsulation is provided (user functions; ADTs)
- objects are persistent.

Compared to Fig. 1, the expressiveness of the various interfaces is as follows:

- the object-oriented interface is (slightly) more powerful than the application-supporting interface
- the object-supporting interface is much higher than the record-level interface, but definitely lower than the former application-supporting interface, since application needs have to be satisfied in the AL.

We have designed the MAD model (Molecule Atom Data model [13]) which provides the properties of the object-supporting interface sketched above. Currently, we are implementing a DBMS kernel PRIMA which offers the MAD model at its interface; an overview of its design and architecture is given in [6]. Hence, we can concentrate on design considerations for the AL in the following.

4. Structure of the application layer

Interactive manipulation of complex engineering objects requires the use of effective communication protocols between kernel and AL as well as a large share of local DBMS processing within the AL in order to guarantee satisfactory response times. On demand, complex objects have to be efficiently extracted and transferred from the *public DB* (managed by the kernel on a server) to the workstation. Then, the AL takes care of these objects—usually for a long time; for temporary storage, it may use a *private DB* on an own disc. To refine the problem, the following questions have to be considered in more detail:

- How does the workstation (and the application program) get its data?
- How does the application program at the workstation manipulate these data?
- How should the changes performed at the workstation be communicated back to the server?
- How should the server database system reflect these changes?

To answer these questions, we introduce the so-called processing model of the AL and some implementation concepts for local buffer management.

4.1. *Processing model of the application layer*

The overall model describing the DBMS activities in the workstation is called the *processing model* of the AL. Its prime purpose is to provide a framework for the exploitation of locality. The examples of Section 2 may convince the reader that locality should be brought closer to the application, even in conventional DBMS applications [17]. Ideally, it is desirable to make a mechanism available that enables the application to reference an object directly, for instance using the pointer concept of a programming language.

With such a typical referencing behaviour in mind, we propose a processing model aimed at high locality of object references. Extraction of data from the public DB is similar to the approach described in [12]. A design transaction issues a *checkout* request if existing design data is needed. Such a request is used to fetch a design object from the public DB. More checkouts may follow when additional data is required by the application. All checked out data is protected by the kernel against concurrent access. The design objects are temporarily stored in the workstation; they are organized in a special main memory structure called *object buffer* which offers fast operational access and a pointer-like reference mechanism. For recovery purposes and for saving particular design states, copies of the design objects may be preserved in the private DB. A design object is committed to the public DB by a *checkin* request. Since commit implies giving up the right of unilateral rollback, the separation of checkin and end of design transaction is meaningless. Hence, we argue for the delay of all checkins to the end of the design transaction.

Summarizing the design transaction, we can identify the following characteristics:

- isolation against concurrent design transactions (provided by the synchronization capabilities of the server DBMS)
- design cooperation only via already checked in (committed) data

- possibly n checkout requests ($n \geq 0$) in combination with no or only one checkin request
- in between there is local manipulation accompanied with the accumulation of design data changes.

Fig. 7 depicts the scheme of such a design transaction following the proposed processing model. After the start of the design transaction it is allowed to checkout the design data needed, using possibly several checkout requests. Then local manipulation is performed on the design objects allocated in the object buffer. It can be structured by issuing one of the following requests:

- SAVE, saving the current design stage;
- RESTORE, backing out to a previously saved design stage;
- SUSPEND, interrupting the manipulation activities (implies a SAVE);
- RESUME, continuing an interrupted design transaction.

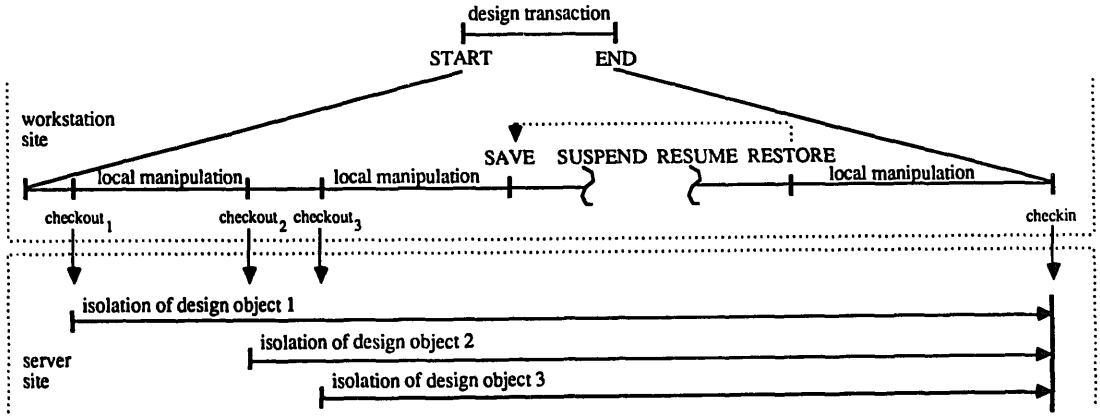


Fig. 7. Sequence of actions in the processing model.

Thus, SAVE and RESTORE provide a user-controlled recovery concept for the design process, i.e. saving a consistent design stage or wiping out the latest actions, while SUSPEND and RESUME support design interruption guaranteeing subsequent processing without loss of information.

Structuring a design transaction with these operations introduce three different states for the transaction, shown in Fig. 8. First, it is simply *unknown*, until the START command is issued. Then it becomes *active*, that means, it is known to the system and it can access and

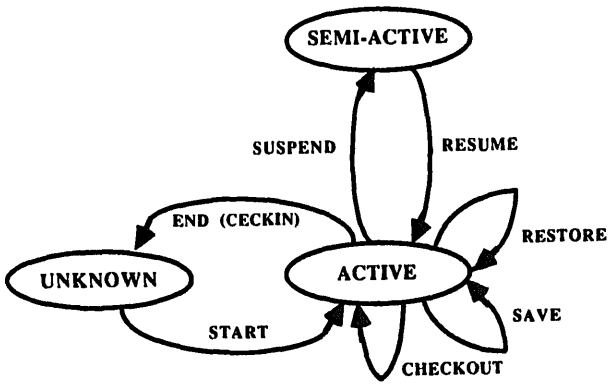


Fig. 8. State diagram reflecting the processing model of applications at a workstation.

manipulate design data. Doing checkout, SAVE, or RESTORE leaves the transaction in the active state, whereas SUSPEND moves it to a *semi-active* state. A semi-active transaction continues to hold all the locks and preserves all its checked out design data in the private DB. Upon a RESUME request, the transaction reenters the active state and finds its specific processing environment as it left it. The final END of the design transaction checks in the newly constructed design objects to the public DB and 'forgets' about the transaction, i.e. turns it to the unknown state.

In the following, we want to describe an adequate implementation concept for the above introduced general processing model (cf. Fig. 7). Obviously we have to be aware of the following optimization criteria:

- minimal number of workstation-server communications
- minimal volume of data transfer
- distribution of the work to do among workstations and server, avoiding duplicated work.

They are supposed to yield high degree of site autonomy and optimized workstation-server cooperation.

4.2. *Implementation of the application layer*

Describing the implementation aspects of our processing model, we first introduce the basic software architecture (Fig. 9). The functionalism of the DBMS kernel interface, which is called object-supporting interface (OSI), is determined by the MAD model. On top of this interface, we have designed a component, called object buffer manager (OBM). The main task of the OBM is local handling and organization of all object-related information needed by the application. Hence, the OBM consists of the preparation component and the object buffer. The preparation component is responsible for fetching and transferring of data from the DBMS kernel to the object buffer and vice versa. The object buffer is a large main memory buffer, that realizes the 'near-by-application locality' and supports the representation of the molecules. In the MAD model, 'molecules' are dynamically defined as sets of 'atoms' (i.e. records, tuples), interconnected by relationships with given semantics. A molecule is supposed to carry all information that describes a design object. A further

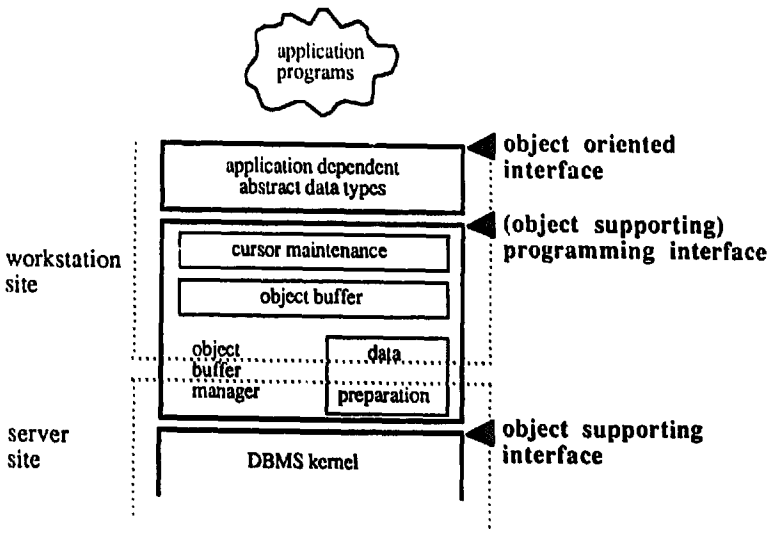


Fig. 9. Components of the application layer.

component of the OBM is the cursor management. Hence, the OBM establishes a powerful data handling interface (object supporting programming interface) at the workstation site. Together with the application dependent program modules it forms the application layer. Fig. 9 shows the basic software architecture; in addition, it indicates their allocation to the associated hardware components. Furthermore, it illustrates that the interface between workstation and server lies inside the OBM layer, that is, our design provides an agent of the OBM at the server site. We assume that this design decision will facilitate all workstation-server crossing operations including checkout and checkin.

After describing the architectural aspects, we now want to characterize the information necessary for workstation-server cooperation in our processing model. First, we have the *query*, defined and later activated with the actual query parameters by a program module in the application layer. The power for query definition is given by the molecule query language (MQL). Second, we have the *answer information* including the query result data aggregated by the DBMS kernel. This information is structured as a set of molecules. Third, there is the *modification information* which comprises all insertions, updates, and deletions made by the application layer. It is encoded as an atom list enhanced by modification flags and specific information about the modification environment. Fig. 10 sketches the level of abstraction for all three kinds of information. It seems to be clear that the molecule set of the answer information is associated with the checkout operation, and the atom list of the modification information corresponds to the checkin operation. Hence, we have a high level of abstraction to formulate the query and to represent the result, and we have a low level to propagate the modifications minimizing the amount of data to be checked in.

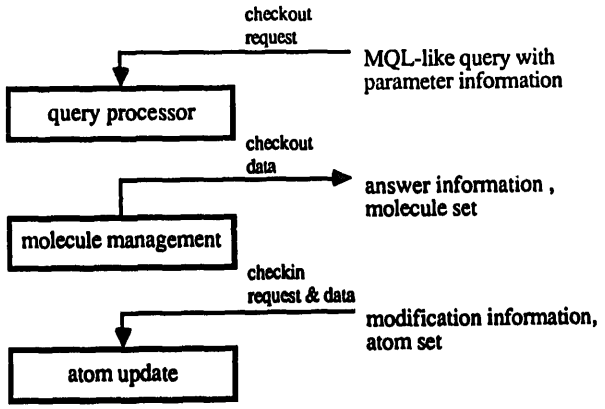


Fig. 10. Data abstraction levels for cooperation.

In the following, we concentrate our discussion on the answer and modification information, because these carry the more interesting issues. Especially, the organization and the internal data structures of the object buffer will be introduced. The answer information consists of a molecule set. Each molecule is composed of a structured set of atoms. Each of them is represented by a list of attributes and is identified by a special attribute, called atom identifier. Molecule identification is done by identification of the root atom. Fig. 11 shows the essential aspects of data structures to represent answer information in the object buffer. The *molecule list* contains the identifier of all molecules constituting the result set of the query. A special hash function h delivers the corresponding root atom index in the *atom table*. The table entry includes some maintenance information. The field modification indicates the type of modification (insert, delete, update). Two separate address fields determine the main memory address of the atom. The area field contains an index of an

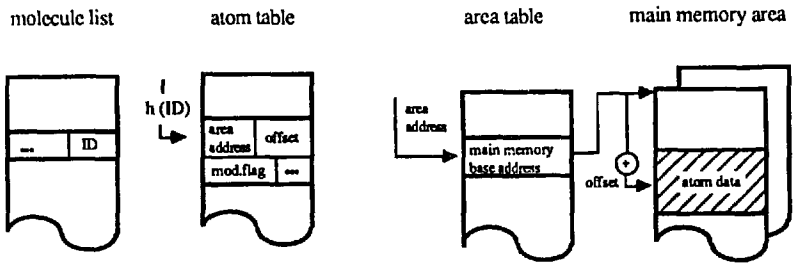


Fig. 11. Organization of the object buffer.

entry in the so-called *area table*, which holds the base address of one *main-memory area*. This base address incremented by the offset field finally yields the atom address within the area. The area concept prevents scattering of the main memory by numerous small atoms and supports the relocatability of the entire molecule set. Relocatability is very useful in such an environment, because the molecule set is frequently moved from the main memory to the private DB and vice versa (SUSPEND, RESUME, SAVE, RESTORE). In this case, relocation is managed quickly by updating only the small area table. The relationships constituting the molecule structure are represented through special reference attributes in the atoms that contain identifiers of other atoms.

The above sketched data structure represents a single result set of just one query. From a logical point of view, it is a snapshot of a database partition. So it seems consequent that an atom is not represented in a redundant manner, if it belongs to more than one molecule within the same result set. On the other hand, an atom is redundantly represented in different result sets. Such multiple occurrences are known to the programs using the object buffer and have to be controlled and managed by them.

The modification information is embedded in a special main-memory area, the so-called *modification area*. It contains all inserted, updated, or deleted atoms of the result set. Therefore, we have no update in place for the first modification of an atom. The updated atom is placed into a modification area, the corresponding modification flag of the atom table is raised, and the addressing fields are adjusted. Then, subsequent modifications of the same atom are executed with an update in place semantics. The modification areas provide some kind of log information at the atom level and are used for propagation of accumulated changes back to the public database (checkin).

The question we want to discuss now is how data in the object buffer is manipulated by the ADTs of the application layer. The atom is the smallest unit of data affected by any modification. We need a *cursor concept* to identify a single atom within the atom set defined by a molecule and within the molecule set given by result set. Such a flat cursor points to only one atom at a time. In principle, it is sufficient for reaching all atoms in the result set, because one can navigate via the reference attributes in the atom data. Nevertheless, the processing characteristics observed in Section 2 have shown that it is useful in many situations to have a more complex cursor, for example a hierarchical one. Often, there are some hierarchical subunits of processing within a molecule. In our implementation, such a hierarchical cursor is defined by a list of atom type names, which marks the paths for the cursor hierarchy, and by identification of the root atom. The concept of hierarchical cursor may be implemented by a hierarchy of dependent flat cursors. Navigation via one cursor automatically affects the subordinate cursors. The idea to support more descriptive (as opposed to procedural) cursor operations is worth more detailed consideration, but it lies beyond the scope of this discussion.

The next question is how all those queries, result sets, molecules, cursors, and atoms are reflected in the programming language which is used to write application-dependent program modules. In principle, there are four different approaches for language binding [11]:

- call interface
- simple host language extension (e.g. CODASYL COBOL-DML)
- embedding database languages in general purpose languages (precompiler, e.g. SQL)
- integrated languages (new data types, e.g. PASCAL/R).

The fourth approach has the advantage that the internal and temporary data is compatible with the external and persistent data, because they have the same logical structure. But, it is not the best procedure from an implementation point of view, because a new language must be designed, a compiler must be written, and so on. So we decided for the third approach designing a host-language embedding using a precompiler.

The use of precompiler statements is sketched in Fig. 12. It depicts a programming scheme of an ADT definition. It includes the declaration and the processing of queries, results, and cursors. The declaration part is transformed into a cursor definition and a declaration of the corresponding internal data structures, e.g. for the atoms constituting the molecules. The data structures are derived from the external schema of the public DB. In the processing part of the ADT, result sets of queries can be assigned to variables of the respective type, the so-called result variables. The assignment causes the activation of the query and thus a checkout, including the binding of program variables to the formal query parameters. Afterwards, the result variables can be read and manipulated with the help of the hierarchical cursors introduced above. The precompiler transforms these operations into accesses to object buffer tables and pointer assignments.

ADT: adt_name

...

declaration of types and variables for molecules, result set of queries and cursors

Initialization

...

assignment of result-set variable (if prefetching is possible) initilizing the ADT

Operation: Operation 1 (... parameters ...)

...

assignment and modification of result sets, molecules, and atom variables using the cursor capabilities to define and process subunits of work

...

Fig. 12. ADT program scheme.

The application dependent ADTs are themselves used at the object-oriented interface by application programs (at a higher level). In addition, this interface offers some general and application-independent operations to organize the designer's activities. For instance, a designer can determine the beginning, the end, the suspension, and the resumption of a design process. Furthermore, he can activate an ADT which makes the corresponding ADT operations available. The related ADT program (cf. Fig. 12) is loaded and its initialization part is executed. Analogously, an ADT can be deactivated thereby giving up the right to further execute ADT operations. But it does not mean that design objects are made available to other designers; note, checkin is postponed until the end of the design process. When a designer finally declares the end of design process, it is assumed that all the modified design objects are to be checked in to the public DB.

5. Implications of server-workstation cooperation on the transaction model

The processing model described in the previous section needs support from a transaction model that copes with various types of failures and with the issues of concurrency on shared data. It has been stated frequently that engineering transactions significantly differ from conventional transactions [1, 9, 10, 12]. They tend to be very long which makes it inadequate to treat failures by rollback to the very beginning and to handle conflicting access to data objects by locks and waits. Instead there should be 'fire-walls' inside a transaction that limit the scope of undoing and provide a starting point after failure.

Additionally, access to design objects that are 'almost complete' could be granted to colleagues working on the same project before the transaction ends. But even in that case designers perform work steps on an object, during which no one could use it, because it is too rough, too incomplete, too fuzzy to be understood. The transaction that belongs to such a work step must therefore remain isolated and must appear atomic to all other designers. It preserves the minimum consistency required even for colleagues. (The degree of consistency is application dependent). This is meant by the term 'design transaction' and will be the key issue of this section.

5.1. The user's view of failures and workstation transactions

A design transaction, although only a small portion of the whole design project, can still be long and needs *recovery points* inside. It consists of a sequence of interactions, i.e. function calls, that may change the state of the system by modifying data. Ideally, creating such a new state should also establish a recovery point. But this may involve significant overhead.

A recovery point is intended to cope with failures. This comprises a wide range from simple operation failure to power reset. In any case the system state is set to the latest recovery point, and the user is informed about the type of failure: A *transient permanent* failure (e.g. deadlock encourages retry of the same operation, whereas in case of a *permanent* failure (e.g. address error in program) this would reproduce the failure. Permanent failures can be bypassed sometimes by another user action. In general, it is necessary to call for the system maintenance.

Not only the system can do wrong. If the designer realizes that his object is not going to satisfy the requirements, he might wish to return to an earlier stage of the design, 'wiping out' anything he has added or changed since then. Defining these stages as well as selecting the one to return to cannot be done by the system. It must be done explicitly by the user and leads to the concept of *savepoints*. Savepoints are often unified with recovery points [4], but

there is no need to do so, and the implementation can be different (imagine a version concept to provide savepoints). The system can use savepoints as recovery points, since the user will know about the related state. But the user cannot use recovery points as savepoints, because he does not know when they are taken.

As the goal is to hide as many failures as possible or at least minimize their effects, there should be much more recovery points than savepoints. Only if it comes out to be too expensive, recovery points will be unified with savepoints. Anyway, it seems appropriate that the SUSPEND command introduced in Section 4 implies the creation of a savepoint.

Even this abstract view of failure and recovery leads to three different concepts in the workstation (Fig. 13):

- the *design transaction* holds the locks on the data (in the server) to provide isolation and preserves minimum consistency
- the *recovery transactions* that are defined by the recovery points and are ideally equivalent to a user operation (a single interaction) in order to minimize loss of work after failures. However, due to performance considerations usually a sequence of operations is secured by a recovery transaction
- *savepoints* serve as a means for user-initiated rollback to reach a previously marked design stage. They confine recovery transactions and may be implemented by a version mechanism, nested transactions [14], or some specialized technique.

From the user's point of view, the property of failure atomicity is only assigned to recovery transactions.

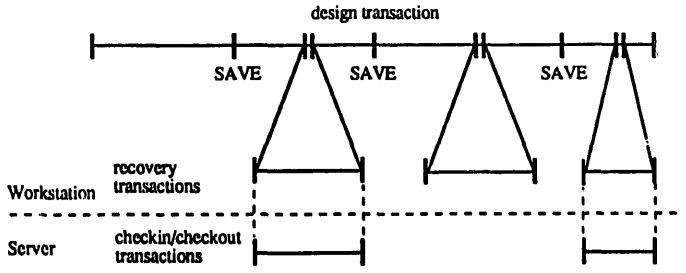


Fig. 13. Transaction nesting in workstation and server derived from requirements at the user's interface.

5.2. Transactions across the workstation-server boundary

The user regards the systems as a whole. On the next level of implementation or refinement there are the workstation and the server cooperation to perform the user's operations. Their mode of interaction has been introduced in Section 4. Along goes a refinement of failures: Workstation and server may fail independently, in each case the failure may concern just one single operation or the whole node, and the failure may be transient as well as permanent. Permanent node failures, e.g. hardware failures, are not discussed in the following, as they need special treatment by the administration (releasing or switching locks in the server to continue on another workstation). The overall goal for the treatment of all other failures is *mutual masking*: An error on the workstation should not bother the server and vice versa. This is only possible if recovery actions in one node do not include UNDO operations in the other node that is still running. How this can be achieved will be discussed for any type of failure.

Before that, a closer look on the private DB in the workstation seems appropriate. The data objects can be in different states, as indicated by Fig. 14. When they are loaded from the public DB of the server, they are supposed to be consistent. And, of course, they are still

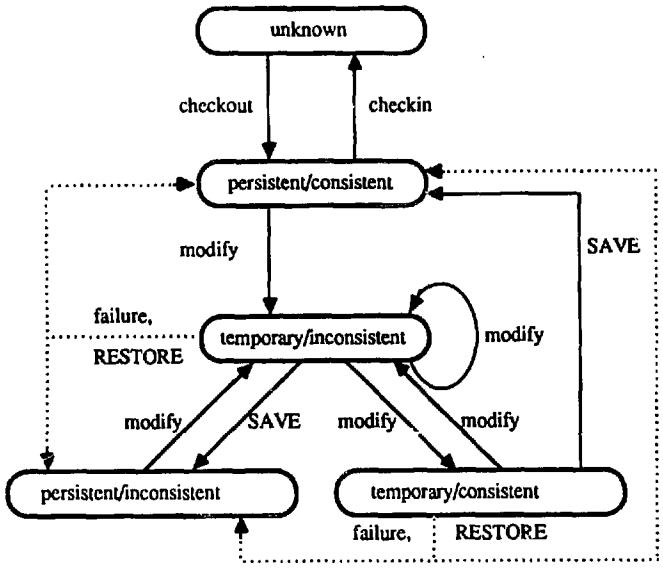


Fig. 14. State diagram for the data in the workstation's private database.

persistent in that they can be loaded again when they are lost due to a failure. Any modification turns them into a state of being temporary and usually inconsistent. Further modifications may be required to reach a new consistent state. But the data remains temporary until a SAVE command is issued to create a savepoint. This can be done with inconsistent data as well as with consistent data.

In the diagram of Fig. 14, data has to be saved before it can be checked in to the public DB. And it has to be consistent, otherwise the checkin will be repelled. A failure on the workstation or the RESTORE command (return to savepoint) put the data back to a persistent state, thereby changing the contents of the data (not shown in Fig. 14). According to the principle of mutual masking, server failures should have no impact at all on thc state of the data. And this implies that

- data checked out is stable in the server
- data checked in does not get lost unless the workstation explicitly requests its UNDO.

All transitions in this diagram (Fig. 14) require a transaction to be active, while a semi-active transaction automatically puts all the data to one of the two persistent states (cf. Section 4.1).

A final remark on Fig. 14: it does not distinguish recovery points from savepoints. If the modify transitions correspond to user operations, the temporary states disappear. The diagram is more general in that it can take into account the internal structure of user operations consisting of several consecutive modifications. Then, every operation ends with an internally generated SAVE defining the end of the recovery transaction.

The concept of workstation-server coupling is based on the strong locality of engineering work. In the context of transactions, this means that most of the recovery transactions will not contain any server calls, i.e. checkout or checkin. They should be managed completely by the workstation without any impact on the server. Of course, this requires a local recovery manager as well as local log files.

The server only knows about the recovery transactions that contain server calls. As indicated in Fig. 13, it regards them as *checkin* or *checkout transactions*. A non-trivial question is whether the server should also know about the context of these transactions, that is, about the savepoints and the design transaction. The alternatives are to be discussed in detail.

5.3. Interaction of workstation and server transactions

The recovery transaction that is executed in the workstation and the corresponding checkin or checkout transactions on the server can be regarded as subtransactions of a single distributed transaction. Particularly, their coordinated completion must be enforced by the two-phase commit protocol (2PC, [4]). Fig. 15 sketches the interaction and shows how failures are treated in the different stages of processing. It is important to notice that in Fig. 15:

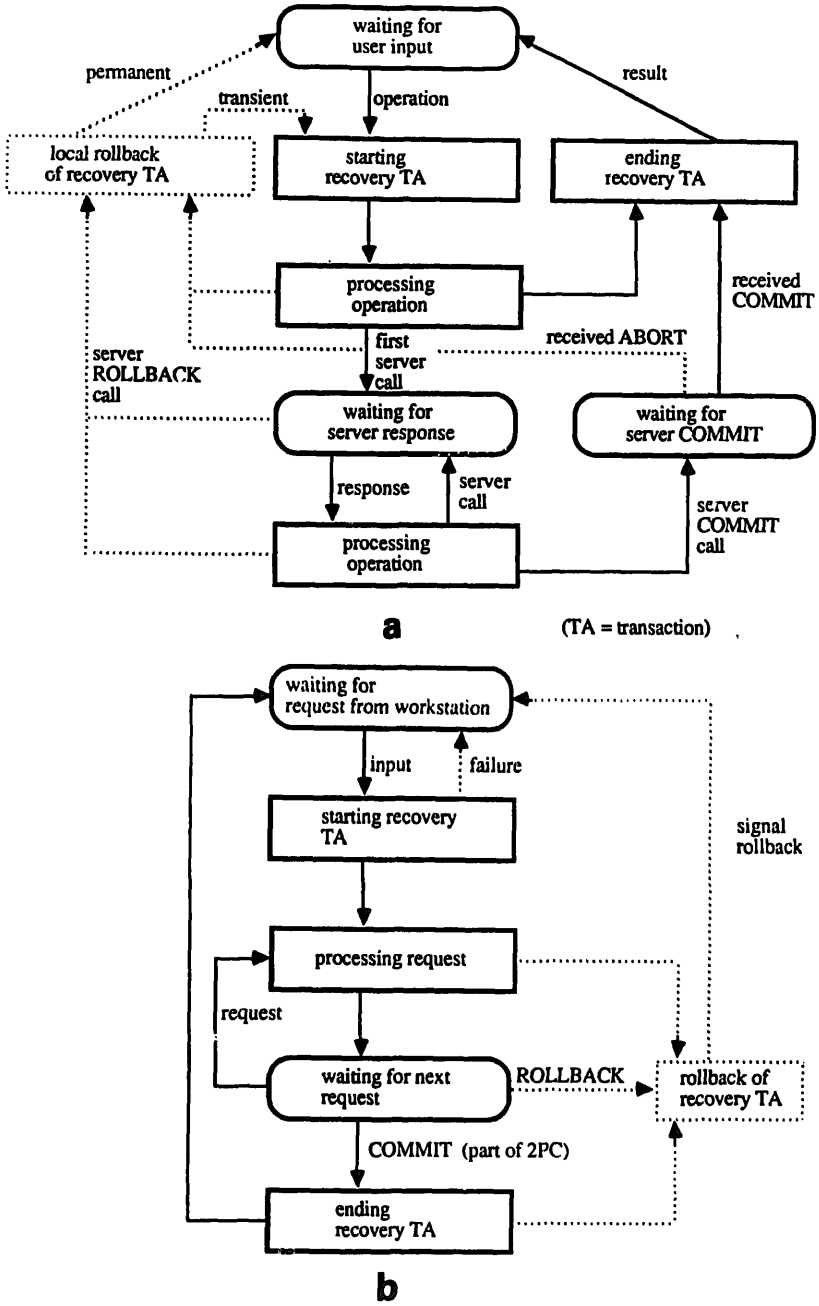


Fig. 15. Interacting state diagrams for workstation and server transaction including failure events:
(a) State diagram for design transaction on workstation
(b) State diagram for recovery transaction on the server.

- every single user operation (a pair of command and response) is assumed to form a recovery transaction, so that transient failures can be handled without the user's noticing
- "starting recovery transaction" therefore includes the logging of the user's input ("operation") to enable restart
- all the rectangular boxes depict local processing, whereas rounded boxes belong to states in which the transaction waits for input or response; these are the points of interaction (cf. state diagrams of [3])
- continuous arrows indicate transitions in normal processing, whereas dotted arrows belong to failures
- the effect of specific user commands introduced in Section 4 (SAVE, RESTORE, SUSPEND) is not shown; RESTORE processing in this environment is cumbrous and must generate compensating server transactions.

This detailed view on the cooperating transactions in workstation and server provides a basis for the discussion of improved transaction concepts.

5.4. Flat or nested transactions in the server

A straightforward approach treats every checkout or checkin transaction as independent and does not take into account the internal structure, i.e. regards it as 'flat'. The first server call makes a recovery transaction on the workstation known to the server and initiates a checkout/checkin transaction. Ending the recovery transaction then includes an additional server call as part of the 2PC. But after that, the server forgets about the checkout/checkin transaction. The consequences of this are:

1 A failure during a recovery transaction that includes server calls—be it on workstation or server—usually involves the other system and requests UNDO on it. The goal of mutual masking is missed.
2 Establishing savepoints (in Fig. 13) and RESTORE processing are very complex operations that comprise transactions and compensating transactions in the server. It can be very expensive. Even if only checkout transactions have been performed since the savepoint, RESTORE processing must notify the server to make it release the related locks. Possibly a good implemantation employs version management on the server DBMS.

The simple approach has a much severer consequence. As anything is committed on the server at the end of the recovery transaction, locks to be held on the design objects until the end of the design transaction cannot be provided by the server DBMS. Instead, a normal data structure (e.g. an OBJECTLOCK relation as in [12]) is used to keep the locks. It must be read by any workstation inside a checkout transaction and must be updated to reflect the granted locks. The term '*application locks*' will be used to refer to this technique. Advantages are:

- failures on the server do not affect the results of committed recovery transactions. Thus the locks survive failures (*persistent* locks).
- semantic knowledge can be assigned to locks. As the data model does not force objects to be disjoint, in many cases the semantic disjointness of objects cannot be derived from the DB schema. Then, the DBMS must control access to all the tuples or atoms in detail which imposes an enormous overhead. The only way to avoid it is exploitation of semantic knowledge.

Disadvantages are:

- locks are not controlled by the DBMS. Access in spite of existing locks is not rejected.

There are some troubles with this concept of simple flat transaction in the server, mostly due to the implementation of SAVE and RESTORE, the necessity of application locks, and the insufficient masking of server failures. It has already been stated that the model of user interaction leads to a nesting of transactions in the workstation. So it is worth investigating whether the concept of nested transaction could be expanded to include the server.

Nested transactions have been introduced by Moss [14]. They have been implemented in a number of experimental systems. A recent article by one of the authors [8] refines the concept by distinguishing the synchronous and asynchronous execution of subtransactions as well as single call and conversational interfaces. It has also shown that savepoints can be used to reduce the transaction UNDO and the amount of work to be repeated after restart. This concept can be applied to the workstation-server configuration.

First, there is a *nesting above the recovery transaction*. If it is maintained by the server as well, this has a number of consequences:

- Locks acquired by recovery transactions are not released at the end of transaction. They are inherited to the parent transaction. If the parent resides on another processor, a local agent is created that represents it. The agent will be discussed in more detail.
- Application locks are no longer needed. More than that, they are completely impossible. Getting an application lock is implemented as an update of a normal data structure (e.g. an atom) that sets a write lock on this piece of data. Since the write lock is not released before the end of the design transaction, other workstations cannot read the lock information—which makes it useless. Hence, no semantic knowledge can be used for locking.
- Implementation of SAVE and RESTORE is much easier, since it could be done by opening a new subtransaction and aborting it, which is both reflected on the server directly. There is no need for compensating transactions.

But what about the persistence of locks? And what is undone by a server failure? To answer these questions, the *agents* have to be investigated more thoroughly. Fig. 16a shows the *dualism of transaction hierarchies* on the workstation and on the server. A recovery transaction up to now is a single transaction that spans both processors, i.e. its end is synchronized by a two-phase commit. The nesting inside a recovery TA is discussed later.

The durability of a recovery transaction is subject to the success of its parent transaction. But it must be durable as long as the parent transaction lives. Therefore, the agent of the parent does not only inherit all the locks, but also carries the UNDO and REDO

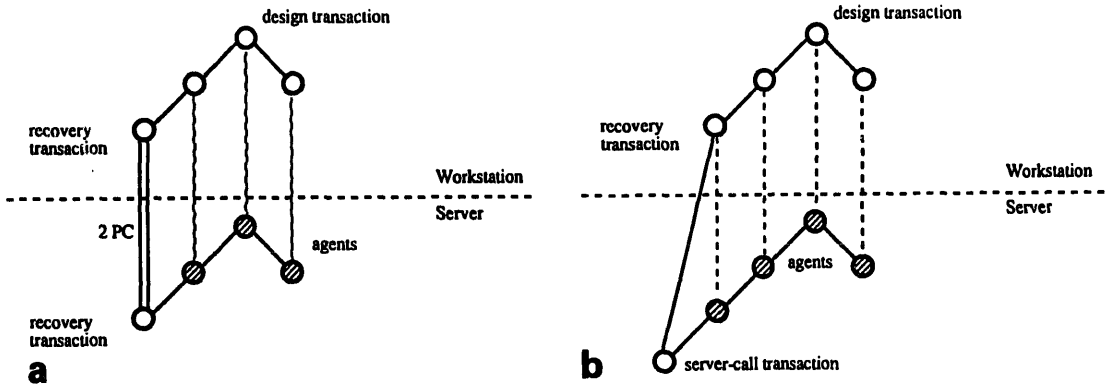


Fig. 16. Dualism of transaction nesting hierarchies on workstation and server:
(a) flat recovery transaction
(b) internally nested recovery transaction.

information. It survives all server failures. In other words, as long as there is no active subtransaction, the agent is always *ready to commit*. In some way, the agent is always in a state comparable to the semi-active state of the design transaction. It guarantees the persistence of locks on the server and limits the scope of recovery to the recovery transaction.

Still the goal of mutual masking is not reached yet, for a recovery transaction also contains a considerable amount of work on the workstation that is destroyed by a server failure. To advance on failure masking, the idea of nested transactions can be utilized inside a recovery transaction as well, making each single server call (usually an MQL statement) a subtransaction. Fig. 16a is then modified as shown in Fig. 16b.

The recovery transaction now itself is represented on the server by an agent with the single call transaction. A server failure causes just the single call to be wiped out. In case of a permanent failure a message is sent to the workstation and the recovery transaction there—still alive!—can decide what to do. The same goes for transient failures, if no input logging is done by the server (could be too expensive). A server failure between two calls has no effect at all on the workstation! Nevertheless, a severe obstacle could be the overhead induced by writing the UNDO and REDO information at the end of every single server call. If this turns out to be too expensive, one has to go back to flat recovery transactions, diminishing the degree of mutual masking of failures. But even then the user need not be involved in the treatment of transient failures.

6. Conclusions

Engineering applications generate huge workloads for DBMS when accurate data models of the design objects are to be maintained by them. Convincing advantages, however, vote for DB-based approaches. Therefore, advanced DBMS should be designed and tailored to specific working environments to support engineering applications and to make interactive designer work feasible. The difficulty of this goal was demonstrated by a number of performance figures derived from realistic prototype applications.

Our design of providing database management services for engineering applications running on dedicated workstations observes the principle of 'near-by-application' locality. Based on the DBMS kernel architecture we have introduced and refined the design of an application-specific DBMS layer with its processing model and implementation. The distribution of DBMS work across server and workstation as well as the particularities of engineering applications call for a transaction model different from business applications. A design transaction exhibiting an internal structure (nesting) was proposed to support long-term designs. Moreover, such a transaction model should include the cooperation among workstation and server thereby mutually masking all failures as far as possible.

Our current implementation of the workstation-server coupling will provide more insight in the various issues discussed. Thus, we hope to demonstrate the feasibility of our approach and, in particular, of the 'near-by-application' locality. Furthermore, we will gain experience with the relevant performance problems of engineering applications.

References

[1] F. Bancilhon, W. Kim and H. Korth. A model for CAD transactions, in: A. Pirotte, Y. Vassiliou (Eds) *Proc. VLDB Conf.* (Stockholm, 1985) 25–33.

[2] P. Dadam et al., A DBMS prototype to support extended NF^2-Relations: an integrated view on flat tables and hierarchies, in: C. Zaniolo (Ed.), *Proc. ACM SIGMOD Conf.* (Washington, D.C., 1986) 356–367.

[3] E. Denert, Specification and design of dialogue systems with state diagrams, in: E. Morlet and D. Ribbens, (Eds) *Proc. Int. Comp. Symp.* (Liege 1977, North-Holland, Amsterdam 1977) 417–427.

[4] J.N. Gray, Notes on database operating systems, operating systems—an advanced course, in: R. Bayer, R.M. Graham and G. Seegmueller (Eds) *Springer Lecture Notes in Computer Science* 60 (1978) 393–481.

[5] T. Härder, C. Hübel, S. Langenfeld and B. Mitschang, KUNICAD—A database system supported geometrical modeling tool for CAD applications (in German), *Informatik Forschung und Entwicklung*, (1) (Springer, Berlin, Heidelberg, 1987) 1–18.

[6] T. Härder, K. Meyer-Wegener, B. Mitschang and A. Sikeler, PRIMA–A DBMS prototype supporting engineering applications, SFB 124, Research Report No. 22/87, Univ. Kaiserslautern, *Proc. 13th Int. Conf. on VLDB* (Brighton, 1987) 433–442.

[7] T. Härder and A. Reuter, Architecture of database systems for non-standard applications (in German), in: A. Blaser and P. Pistor (Eds) *Database Systems for Office, Engineering, and Science Environments* (Springer, Berlin, Heidelberg, 1985) 253–286.

[8] T. Härder and K. Rothermel, Concepts for transaction recovery in nested transactions, in: U. Dayal and I. Traiger (Eds) *Proc. ACM SIGMOD Conf.* (San Francisco, 1987) 239–248.

[9] R. Katz and S. Weiss, Design transaction management, in: *Proc. 19th Design Automation Conf.* (June 1983).

[10] W. Kim, R. Lorie, D. McNabb and W. Plouffe, Nested transactions for engineering design databases, in: U. Dayal, G. Schlageter and L.H. Seng (Eds) *Proc. VLDB Conf.* (Singapore, 1984) 355–362.

[11] M. Lacroix and A. Pirotte, Comparison of database interfaces for application programming, in: *Inf. Systems*, 8 (3) (1983) 217–229.

[12] R. Lorie and W. Plouffe, Complex objects and their use in design transactions, in: *Proc. Data Base Week: Engineering Design Applications* (San Jose, 1983) 115–121.

[13] B. Mitschang, MAD—A data model for the kernel of a non-standard database system (in German), in: A. Blaser and P. Pistor (Eds) *Database Systems for Office, Engineering, and Science Environments* (Springer, Berlin, Heidelberg, 1987) 180–195.

[14] J.E.B. Moss, Nested transactions: an approach to reliable computing, M.I.T. Report MIT-LCS-TR-260, M.I.T., Laboratory of Computer Science (1981).

[15] H.-B. Paul, H.-J. Schek, M.H. Scholl, G. Weikum and U. Deppisch, Architecture and implementation of the Darmstadt database kernel system, in: U. Dayal and I. Traiger (Eds) *Proc. ACM SIGMOD Conf.* (San Francisco, 1987) 196–207.

[16] A.A.G. Requicha and H.B. Voelcker, Solid modelling: a historical summary and contemporary assessment, in: *IEEE COMPUTER Graphics and Applications*, 2 (2) (March, 1982) 9–24.

[17] M. Stonebraker and L. Rowe, Database portals—a new application program interface, in: U. Dayal, G. Schlageter and L.H. Seng (Eds) *Proc. VLDB Conf.* (Singapore, 1984) 3–13.