5066-6

Database Recovery in the Design Environment: Requirements Analysis and Performance Evaluation

Zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

von der Fakultät für Informatik

der Universität Karlsruhe (Technische Hochschule)

genehmigte

Dissertation

von

Cirano Iochpe

Mestre em Ciência da Computação

aus Porto Alegre, Brasilien

Tag der mündlichen Prüfung:	12.12.1989
Erster Gutachter:	Prof. Dr.-Ing. Peter Lockemann
Zweiter Gutachter:	Prof. Dr.-Ing. Andreas Reuter

Gedruckt mit Unterstützung des Deutschen Akademischen Austauschdienstes (DAAD)

À "Pequena",
pelo grande apoio, dedicação e entusiasmo

Acknowledgements

The present work has been developed as part of a Ph.D. course at the Computer Science Department of the University of Karlsruhe. It has been supported by the Deutscher Akademischer Austauschdienst (DAAD), to which I am very grateful.

I would like to especially thank Prof. Dr. P. Lockemann for the chance he gave me to carry out this work as a member of his group of researchers. He has always encouraged me and his comments on my work were very helpful. I am also indebted to Prof. Dr. A. Reuter for the fruitful discussions and helpful advices on the basis of which many good results were achieved.

I would also like to thank my colleague Günter von Bültzingsloewen for the productive discussions and the support he gave me throughout my stay at the Forschungszentrum Informatik (FZI) in Karlsruhe. I am also very grateful to Angelika Kotz, Ralf Kramer, Peter Liedtke, Michael Schryro, Martin Haas, Herbert Schickle, and Guido Moerkotte for the good ideas they always gave about the work. I am thankful to Nelson M. Mattos, Taísy Weber, Raul Weber, and Lia G. Golendziner for their motivation and (long distance) support.

I am indebted to the students Holger Schmid, Andreas Tenginsky, Holger Diehl, Mathias Küster, and Wolfgang Weber who helped to implement the whole simulation system which constitutes a basic tool for the development of my work. I would also like to thank the secretaries Helga Gerstner, Eva Mai, Beatrice Kuch, and Marlis Heilmann for their decisive support to the realization of this work.

I would like to especially thank my wife Eliete for her support and the great patience she demonstrated during these almost five years in West Germany. Finally, I am deeply indebted to my parents who showed me the way and always gave me the motivation and support to follow it.

Abstract

In the past few years, considerable research effort has been spent on data models, processing models, and system architectures for supporting advanced applications like CAD/CAM, software engineering, image processing, and knowledge management. These so-called non-standard applications pose new requirements on database systems. Conventional database systems (i.e. database systems constructed to support business-related applications) either cope with the new requirements only in an unsatisfactory way or do not cope with them at all. Examples of such new requirements are the need of more powerful data models which enable the definition as well as manipulation of fairly structured data objects and the requirement of new processing models which better support long-time data manipulation as well as allow database system users to exchange non-committed results.

To better support new data and processing models, new database systems have been proposed and developed which realize object-oriented data models that in turn support the definition and operation of both complex object structures and object behavior. In design environments as the ones represented by CAD applications, these so-called non-standard database systems are usually distributed over server-workstations computer configurations. While actual object versions are kept in the so-called public database on server, designers create new objects as well as new object versions in their private databases which are maintained by the system at the workstations. Besides that, many new design database system prototypes realize a hierarchy of system buffers to accelerate data processing at the system´s application level. While the storage subsystem implements the traditional page/segment buffer to reduce the number of I/O-operations between main memory and disk, data objects are processed by application programs at the workstation at higher levels of abstraction and the objects are kept there by so-called object-oriented buffer managers in special main memory representations.

The present dissertation reports on the investigation of database recovery requirements and database recovery performance in design environments. The term design environment is used here to characterize those data processing environments which support so-called design applications (e.g. CAD/CAM, software engineering). The dissertation begins by analyzing the common architectural characteristics of a set of new design database system prototypes. After proposing a reference architecture for those systems, we investigate the properties of a set of well known design processing models which can be found in the literature. Relying on both the reference architecture and the characteristics of design processing models, the dissertation presents a thorough study of recovery requirements in the design environment. Then, the possibility of adapting existing recovery techniques to maintain system reliability in design database systems is investigated. Finally, the dissertation reports on a recovery performance evaluation involving several existing as well as new recovery mechanisms. The simulation model used in the performance analysis is described and the simulation results are presented.

Contents

List of Figures

Chapter 1

Introduction

1.1 Database Systems for Business-Related Applications

Database system technology has basically emerged to support computer system applications which require data to be shared among different programs. The concept of sharing data was first introduced in the mid 1950's through the common-areas provided by Fortran to enable communication among different programs. In the past thirty years, new concepts have been proposed as well as new mechanisms have been designed and implemented to better support data sharing activities.

The need of shared databases (DBs) has first been felt by the implementors of business-related applications. This fact can in part explain why database technology evolved towards the development of database systems (DBSs) which are mainly concerned with the requirements of these types of applications. Since business-oriented database systems represent the majority of the existing DBSs, they are usually called conventional or standard database systems. These adjectives distinguish business-oriented DBSs from other database systems which have been designed to support other types of applications (e.g. design applications like CAD/CAM and software engineering). The latter are usually referred to as being non-standard (or non-conventional) DBSs.

The basic idea underlying database systems is to provide a computer system with the capacity of both managing large amounts of persistent data in a reliable way and permitting these data to be shared among various users. During the last thirty years, researchers at both industry and university have developed concepts and implemented systems to support this idea. Today's DBSs rely on three basic abstractions: **data independence**, **transactions**, and **location independence** [BrMa88]. These abstractions form the conceptual basis for the implementation and management of large shared databases.

Data independence represents the capacity of database systems for hiding database implementation details from the application developers. Data independence ideally guarantees that changes to the database (e.g. the introduction of new data types or the deletion of existing access paths) do not affect existing application programs. Similarly, modifications on existing applications as well as the addition of new applications to the system should not alter the functionality of the database.

Database systems usually realize the data independence abstraction on the basis of a stepwise refinement of the database design, that is, on the basis of a stepwise process of capturing and representing the application's characteristics in the database. One can subdivide the database design process into three main steps: conceptual design, logical design, and physical design. Each database design step further refines the description of the application to the DBS made by means of earlier design steps until the details related to the physical implementation of the database (e.g. data structures and access paths) can be derived from the static (e.g. data types) and dynamic (e.g. transactions) properties of the application being designed. By both requiring that the development of the application programs rely on early design steps (i.e. those design steps which do not model the physical implementation of the database) and realizing the physical representation of the database on the basis of later design steps, the database system hides the details of the physical implementation of the database from the application. In this way, data independence is achieved.

The conceptual design is the process of representing the main application properties on the basis of a conceptual data model (e.g. Entity-Relationship data model). The results of the conceptual design is a formal representation of the properties of the application. This representation is related to no particular database system, though. It can therefore be mapped onto the specific logical model supported by the target machine.

The logical design translates the results of the conceptual design into a logical one based on a specific logical data model such as the relational, hierarchical, or network data model. Standard database systems support only one of these logical models. Logical data models represent a framework for the description of data entities and relationships among data. Besides of that, these models define sets of high-level data definition and data manipulation operations on the basis of which the application programs are developed. Logical data models are not concerned with the physical implementation of the database, that is, with the definition and management of data structures and access paths for both storage and manipulation of data.

Finally, the logical design is translated into the physical database design which describes the implementation details of the database on the target machine. Through the physical design, the database designer specifies the data structures where the database should be stored and the access paths through which the data are to be retrieved.

The **transaction paradigm** assures the isolation of concurrent work on the database as well as the recoverability of the database state in case of failures. In this way, the transaction concept constitutes the basis for the support of data sharing and system reliability in database systems.

The transaction represents a logical unit of work that includes several properties [HäRe83]. It is supposed to be always **correct**. That is, transactions which update the database always bring it from one consistent state into another consistent state. The transaction is **atomic** in that either all its statements are successfully processed or none of its effects on the database survives. Moreover, the transaction´s work is **isolated** from the work of other transactions which execute in parallel to it. That is, every running transaction can perceive only the results of already committed (i.e. successfully terminated) transactions. Finally, the results of committed transactions are guaranteed to survive system failures (i.e. they are **persistent**).

The transaction property of isolation forms the basis for data sharing in database systems. Since transactions access only data which have been inserted or updated by committed transactions, and committed transactions are, by definition, always correct, transactions access only consistent data even when they execute in parallel to other transactions.

Most standard database systems realize transaction isolation on the basis of the serializability theory [BeHG87]. The basic idea of this theory is to resolve data access conflicts among transactions which execute in parallel by producing serializable schedules. A schedule is serializable if and only if the results produced by the transactions participating in it are equivalent to the results which can be obtained by some serial execution of these transactions. Since every transaction is considered to produce a consistent database state when it runs alone in the system, a serial schedule of transactions always produces a consistent database state. Since each serializable schedule is equivalent to, at least, one serial schedule, serializable schedules always produce consistent database states, too.

The data reliability property of database systems relies on the transaction´s atomicity and persistency properties. The atomicity property guarantees that none of the transaction´s temporary results survives in the database in case of transaction or system failures. The survival of results of committed transactions in case of system failure is enforced by the persistency property.

The third abstraction on which database systems rely, namely **location independence**, permits applications to access data in distributed systems without being concerned with the location of the data and data repositories being used.

1.1.1 Some Important Properties of Business-Related Applications

In 1969, the CODASYL Database Task Group (DBTG) published the specification of the first formal database model for business applications [CODA71]. It is a network data model designed to cope with the processing requirements of business applications. From the characteristics of this model, one can identify the main requirements posed by this kind of applications. Network as well as hierarchical data models are mainly designed for the iterative processing of lists of data. Their basic constructs are entities and relationships (i.e. 1:N by the hierarchical model and N:M by the network data model). These models support a record-at-a-time processing mode. Application programs navigate within entities from one entity instance to the other. By following relationships among entity instances, the programs can also access entity instances of oder entities. Entity instances (and some times relationship instances too) are represented as data records with attributes.

Although the DBTG´s data model has been published in the late 1960´s, this group had already been working on it for ten years. During this time, many new database systems were developed which supported either a network or a hierarchical data model. At the end of the same decade, Ted Codd of IBM developed the basis of the relational data model which was to become a standard for the database systems of the 1980´s [Codd70].

The concept of a relation (i.e. a set of flat data records of the same type) together with set theory, and predicate calculus form the basis of the relational data model. In this model both entities and relationships are represented as relations. The relational model supports N:M relationships and presents a set-at-a-time processing mode. Nevertheless, most applications based on the relational model continue to process data in a record-at-a-time basis, because their application programs are written in some procedural, general-purpose programming language (e.g. COBOL). In this case, the relational queries and update statements are embedded in these programs. This induces what François Bancilhon calls the impedance mismatch [Banc88]. Although the relational statements produce relations (i.e. sets of records) as a result, the application programs must process these relations in a tuple-at-a-time basis.

The data processing activity of business-related applications is characterized by short-duration transactions in batch systems and short-duration dialogue steps in on-line systems. Typically, transactions (or dialogue steps) execute in some seconds. While batch transactions may have to process large volumes of data, dialogue steps usually manipulate only a few data records. Anyhow, only a few short-duration operations are applied to each data record or tuple processed. Transactions as well as dialogue steps show high locality of data access, that is, most of them access practically the same parts of the database.

The number of concurrent transactions in business-oriented database systems can vary depending on the application being supported [Meye86]. While a throughput of less than one transaction per second can be tolerated by many small dialog systems (i.e. DBSs connected to five or ten terminals), there exist some applications (e.g. flight reservation systems) which require a throughput of more than two hundred transactions per second.

1.1.2 Some Architectural Characteristics of Standard Database Systems

Standard database systems are realized as either centralized or distributed computer systems. Centralized database systems typically support either multi-user or single-user processing modes. Distributed systems mainly realize multiprocessing environments.

In centralized systems, the whole database resides in one single processing node. In distributed systems, the database can be distributed over various nodes, though. Moreover, the whole database or parts of it can be replicated in distributed database systems.

In the case of a centralized system, transactions execute at the only processing node. In distributed systems, a transaction at one node can spawn (sub)transactions at other nodes to access data residing there. In all cases, though, (sub)transactions executing at one processing node directly access only the data residing on that node. Since transactions may have to process relatively large volumes of data for short periods of time, it is more efficient to start sub-transactions at the nodes where the desired data reside than transfer them to the processing node where the original transaction has been started.

The architecture of database systems can be viewed as a hierarchy of hardware and software layers [LoSc88]. Each layer of the architecture accesses data in some pre-defined representation form at the next-lower layer´s interface (the lowest layer directly accesses data from the stable storage device), realizes a higher data abstraction, and implements operations on it which will be called by the next-higher system layer. In this way, the system realizes relatively complex data models (i.e. specific data abstractions together with the operations on them) step-by-step on the basis of very simple data representation forms (e.g. bits on disk).

DBS users (e.g. persons at computer terminals, application programs) communicate with the database system through one or more of its layer interfaces. Figure 1.1 shows a general architecture for standard database systems which implement the relational model as it is presented in [LoSc88]. Although each layer interface supports a different data abstraction (i.e. data model), standard database systems physically store data in main memory using only one data representation form, namely data pages or data segments. Higher data abstractions are realized on the basis of the data stored on pages every time operations which use them are executed by the respective layers (i.e. the layers which implement those data abstractions). Thus, although the architecture of standard database systems is layered, these systems do not really store different data representation forms in main memory. Although most standard DBSs store database operation results in form of single data records or tuple sets in the application program´s work space in main memory, this area cannot be considered part of the database system. Its contents can neither be protected by the DBS nor used by it in internal activities.

The implementation of the database buffer (i.e. the main memory repository for the database) at only one layer of the system is usually done for efficiency reasons. The time to map one data representation form into another takes relatively long when compared to the time needed by the transaction to process the data whose representation form is being changed.

1.2 Database Systems for Non-Standard Applications

In the last few years, considerable research effort has been spent on making database systems suitable for supporting new, more challenging applications. These so-called non-standard applications consist in a large set of non-business-oriented applications which pose new requirements on database system support [Lock85]. Some examples of non-standard applications are artificial intelligence (AI), office automation, and engineering applications, as computer aided design (CAD), computer aided manufacturing (CAM), and computer aided software engineering (CASE). The actual investigation of non-standard database systems aims at extending database technology to better support these new applications.

Although standard DBS technology represents a solution for the problem of managing large amounts of data in a shared and reliable way for non-standard applications, too, this technology cannot cope well with other requirements of those applications. On the one hand, applications such as CAD/CAM, office automation, CASE, or knowledge representation need more powerful data models which support the definition and manipulation of new types of data that are not supported by classical data models. Examples of such data types are geometric data which are used in engineering design and cartography applications, imagery, voice, AI knowledge representation, text, and signals of various types. On the other hand, some non-standard applications require new database processing models that are not supported by standard database systems. For instance, design transactions take typically much longer than (conventional) business transactions (e.g. days or weeks). Moreover, they may need to process non-committed data, that is, data which have been either created or updated by transactions whose execution has not yet terminated.

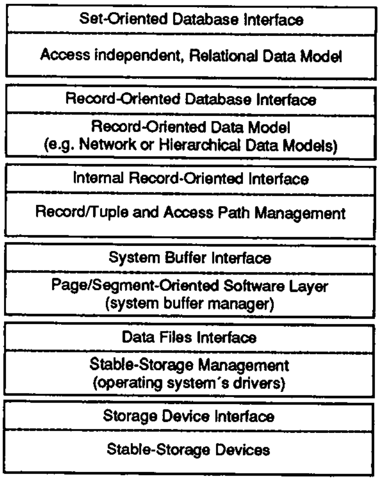

Set-Oriented Database Interface
Access independent, Relational Data Model
Record-Oriented Database Interface
Record-Oriented Data Model
(e.g. Network or Hierarchical Data Models)
Internal Record-Oriented Interface
Record/Tuple and Access Path Management
System Buffer Interface
Page/Segment-Oriented Software Layer
(system buffer manager)
Data Files Interface
Stable-Storage Management
(operating system's drivers)
Storage Device Interface
Stable-Storage Devices

Fig. 1.1: A general architecture for standard database systems

The data objects to be defined and manipulated by many non-standard applications (e.g. AI applications, design applications) present a more complex structure than those processed by business-related applications. Most of them are both hierarchically structured and composed of a number of (sub)objects which can, in turn, be composite objects, too. Moreover, in some applications it is possible to find two or more objects that, in turn, have one ore more common (sub)objects. As a consequence of that, the database for these applications can be represented by a directed acyclic graph (DAG). Besides, to efficiently support the manipulation of highly structured data objects, the DBS should be able to identify them uniquely and to provide database operations which can manipulate them as a whole (i.e. together with their subobjects). Furthermore, these operations should be tailored to the needs of the specific application being supported. While the hierarchical and the relational data models cannot represent the new types of data objects in a natural way, the network data model do not support efficient operations on them. Consequently, new data models have been proposed which try to better capture the specific properties of non-standard applications (e.g. [DAM86a], [LeRV87], [Mits87], [Pist86], [RoSt87]).

Most of the new data models proposed are object-oriented. This type of data models is based on the object concept of object-oriented programming languages. Database systems which realize object-oriented data models are called object-oriented database systems (OODBSs). This type of DBS should integrate the properties of standard database systems (e.g. persistency, concurrency, recovery) with the concepts of object-oriented programming languages [OODS89]. For our discussion, the most important of these concepts are complex objects, object identity, encapsulation, types or classes, and inheritance. Depending on which object-oriented concepts the DBS realizes, OODBSs can be subdivided into three categories [Ditt89]. The DBS is said to be structurally object-oriented if it implements only the structural features of objects-oriented programming languages (i.e. complex objects and object identity). On the other hand, the DBS is called behaviorally object-oriented if it implements object behavior on the basis of classical data models (which do not support structured objects). Concepts related to object behavior are, for instance, encapsulation, types or classes, and inheritance. Besides increasing data independence, both object encapsulation and types or classes enable the dynamic expansion of the data model through the definition of new data types and new operations. Database designers can accomodate the requirements of new applications to existing database systems by defining data types and data operations which better capture the static and dynamic properties of those applications. The inheritance property of object-oriented systems enables the definition of hierarchies of data types (or classes) in the database where subtypes can inherit common attributes (i.e. characteristics) and methods (i.e. operations) from their ancestor types. Database systems which implement structural as well as behavioral object concepts are called full object-oriented.

To better support new data and processing models, new DBS architectures have been proposed. Some researchers expanded already existing database systems to allow object-oriented processing activities (e.g. [HaLo81]). Others expanded existing data models and implement them on the basis of fairly new system architectures (e.g. [StRo86]). A third group works on the basis of kernel database system architectures which shall provide basic database support (e.g. data management, concurrency control, recovery) and realize a simpler object-oriented data model (e.g. a structurally object-oriented data model). On top of this architecture, relying on the kernel´s interface, the system realizes an additional layer (i.e. the application-oriented DBS layer) which copes with the special needs of the application being supported (e.g. [DiKM85], [HMMS87], [KeWa87], [Paul87]).

Although the term non-standard applications can efficiently distinguish the group of non-business-related applications from the group of the business-related ones, this term can also lead to the erroneous conclusion that all non-standard applications have similar dynamic and static properties. For instance, opposed to many design applications image processing applications do not require new data processing models. Although image processing activities may take longer than (conventional) business-oriented transactions, they should not take days or weeks, as design transactions do. In the following, we concentrate on the discussion of engineering design applications. We mainly do that because most of the object-oriented data models proposed as well as most of the non-standard DBS prototypes to be investigated in this work were designed to cope with this type of applications.

1.2.1 Some Important Characteristics of Design Applications

Design applications deal with the planning, development, and relization of technical systems. Hence, these applications are usually related to engineering activities. Examples of design applications are the development of software systems, the design of mechanical parts, the planning of electronical circuits, etc. Because of the ever growing extension and complexity of design activities, computer systems have been developed to support such applications. Most of the existing computer aided design systems consist of file servers and software tools (e.g. graphic editors, mask compilers) which help the designer by projecting, realizing, and testing his design work.

Concerning the distribution of tasks and personnel, the overall design enterprise presents a hierarchical structure [BaKK85]. Normally, larger projects are subdivided into subprojects that can, in turn, be further partitioned until the individual design subtasks achieve the expected grade of complexity and independence from each other. The resulting design subtasks are then executed by groups of designers. In every group, each designer executes a part of the group´s work. Reflecting the hierarchical partitioning of the overall design work, designers of different groups manipulate different design objects. In most cases, designers of different groups access common data only for read purposes [Kelt88]. The same occurs when a designer of one group needs to access private data of another group.

Designers working on the same group access the same data objects, though. Contrary to users in business-related applications that access data concurrently, designers belonging to the same group tend to cooperate during their data processing activities. That is, design transactions may have to see results of other, not yet committed design transactions to continue activities. As a consequence, design database systems should relax the transaction property of isolation. The relaxation of concurrency control rules should be done in a controlled way, though. Korth [KoKB87], for instance, proposes that concurrency control should be exercised on an object basis and not on a transaction basis. That is, a transaction should be allowed to release the locks it holds on an object even before it commits, as long as the transaction or the user executing it decides that the processed object is consistent enough to be seen by other transactions/users.

Usually, designers belonging to the same group work in physical contiguity (e.g. in the same room or building). Therefore, they can easily get in touch and discuss next steps of their common design work. Designers discuss the design work also in meetings and by means of electronic mail. The physical distribution of design groups and designers as well as the hierarchical structure of the overall design enterprise have influence on the configuration of computer aided design systems and, consequently, on the architecture of database systems supporting the design environment.

1.2.2 The Computer System Supporting the Design Environment

Usually, the computer system supporting the design environment is organized as a server-workstation computer network. Figure 1.2 ilustrates this kind of system architecture. The processing nodes are connected through a local area network (communications subsystem) which allows direct communication among them.

Server processing nodes work as data managers and control the data and software tools which are global to the design enterprise. For some processing models, these data constitute the so-called public database. Server nodes may themselves be distributed computer systems but are treated as single entities by other processing nodes.

Designers process objects at the workstations which are usually located at their offices. The workstations are based on relatively efficient CPUs (e.g. 32-bit microprocessors), are equipped with main memory of several megabytes and a monitor which, usually, can support graphic applications. For the (near) future, it is expected that most workstations will be equipped with private disk units. At the workstations, disks are used to store the designer's private data, that is, non-committed design data (e.g. non-released design object versions). For some processing models, these data constitute the so-called designer's private database.

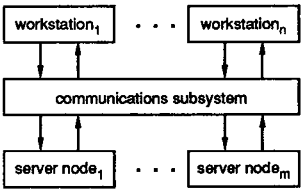


Fig. 1.2: Design environment's computer configuration

1.2.3 Database Systems for the Design Environment

As already mentioned, many computer-aided design systems have already been implemented and put on the market. Most of them are based on file servers, though, which do not offer all the advantages of a database system, namely data independence, concurrency control, recovery, etc. Other systems have been realized on the basis of standard DBSs which support only conventional data models. Since these systems do not cope well with some design application requirements on data and processing modes, researchers are presently investigating new DBS technologies to better support the design environment.

No matter what type of DBS is going to be integrated into the design environment, this system will have to be a distributed one. The so-called public database system will be implemented at the server node(s). It will manage released data object versions and other information concerning the overall design enterprise. At every workstation, a local DBS will manage the designers' work on their private data. Private databases will be managed by local DBSs at workstations owning disks. Through the communications subsystem, the various DBSs of the overall system will exchange data from one database into the other.

New data models for design applications support the definition and manipulation of highly structured data objects (i.e. complex or molecular objects as they are defined in [HaLo81] and [BaBu84]). Database systems supporting the design environment will use objects as the unit of data transfer for the communication between computer nodes.

In contrast to database systems for business-related applications, DBSs for design applications will not always process data at the site where they are located. Since the designer at the workstation can process data objects for very long periods of time (i.e. days, weeks), it is more efficient for the DBS to transfer data from the server to the workstation instead of having to spawn sub-transactions at the server node to process objects there on behalf of the designer at the workstation. Local DBSs will copy data objects from the public database into the designer's private database at the workstation by executing so-called CHECKOUT operations [HaLo81]. To transfer objects from the private database back into the public database, the DBS will execute so-called CHECKIN operations. To distinguish CHECKIN operations which really update old object versions in the public database from those which only release locks in that database, a so-called UNCHECKOUT operation has been introduced [DAMO88a].

Through CHECKOUT operations as well as CHECKIN operations and UNCHECKOUT operations, the server DBS can control concurrent access to the public data object versions. Most of the design processing models proposed differ from each other in when exactly these operations may be issued (e.g. [HaLo81], [LoPl83]).

A number of architectures for non-standard database systems have recently been proposed and can be found in the literature (e.g. [HäRe85], [Depp86], [DiGl87], [HMMS87], [Paul87]). Most of these proposals rely on the architectures for standard database systems presented in [Senk73] and [HäRe83].

Figure 1.3 presents a somewhat modified version of the non-standard DBS architecture presented in [Depp86]. The highest system layer (i.e. the application layer) realizes the application-oriented data model which can be thought of as a complex, object-oriented data model. This model is implemented on the basis of both objects and operations of the next-lower layer's interface. The complex object manager, in turn, realizes a more general and less complex object-oriented data model. This so-called internal object model usually constitutes the uppermost interface of the DBS kernel in design database systems (e.g. [HHMM88]).

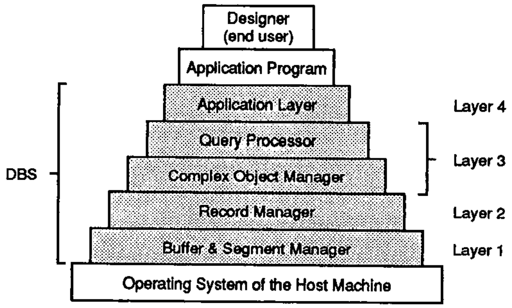


Fig. 1.3: A layered architecture for non-standard database systems

The complex object manager maps complex objects onto sets of tuples or records at the interface of its next-lower layer, namely the record manager. This system module uses the page/segment-oriented interface realized by the buffer-and-segment manager to extract records from and insert records onto data pages. The buffer-and-segment manager maps pages and segments onto files at the operating system's file interface.

The segment layer can also implement the concept of physical clusters (i.e. segments). A segment can be viewed as a set of independent data pages that can be referenced as a single entity (i.e. a data cluster). Physical clusters are used to accelerate the transfer of complex objects between main memory and stable storage.

Design database systems can be either especially designed to support a specific application (i.e. tailored design DBS) or developed as kernel design database systems. Since tailored DBSs need, respectively, to support only one application (or, at most, a group of similar applications), the requirements posed on these systems are then well known to the DBS designers. Hence, tailored system architectures can be optimized to guarantee system efficiency at run time. For instance, in principle tailored design DBSs need not realize the complex object manager as it has been explained above. Since only one application-oriented data model is to be implemented, it could be directly constructed on the basis of the record manager. Another possible system design optimization could be achieved by letting the application layer directly determine which data should be kept together in the same segment.

Although tailored design DBSs can be more efficient than kernel design DBSs, they must be completely redesigned and reimplemented every time a new application must be supported. The concept of a design DBS kernel represents an alternative to tailored systems. Kernel design DBSs can be viewed as consisting of two main parts: the kernel subsystem and the application subsystem. The kernel realizes a more general and less complex, object-oriented data model (through the complex object layer). On the basis of this data model, the application subsystem implements a data model which can better capture the specific semantics of the application being supported. If new applications are to be supported by the kernel design DBS, only the application subsystem must be changed. The kernel part of the system remains the same. Moreover, the same DBS kernel can support different application types at the same time.

As a consequence of providing various external interfaces (e.g. kernel interface, application-oriented interface), kernel design DBSs need to support efficient main memory access at different levels of abstraction (e.g. page/segment level, object level). Many non-standard DBS projects have already identified this necessity of kernel systems and try to solve the problem by realizing what we call hierarchies of system buffers (e.g. [StRo84], [Rowe86], [DePS86], [KeWa87], [HHMM88]). As opposed to conventional database systems which usually realize only one buffer manager at the system's page level, these new systems implement, at least, two buffer managers at different levels of abstraction (e.g. page level and application-oriented level). System buffers at lower levels help improving the performance of the kernel, while the buffers at higher abstraction levels support the access patterns of the application. The concept of a hierarchy of buffers can be explained by means of Figure 1.4 which shows a somewhat modified version of the design DBS architecture presented in [HHMM88].

Figure 1.4 depicts the architecture of a kernel design DBS which is distributed over a server-workstation architecture. The kernel at the server node actually works as a data object server. It supplies data objects to the local data systems at the workstations and integrates updated data objects back into the public database. The kernel consists of two main layers: the object/tuple-supporting layer (L2) and the page/cluster-supporting (L1) layer. The local data systems at the workstations consist of two layers each: the application layer (AL) and the object-supporting layer (OSL).

The page/segment buffer of the system is located in L1. For every CHECKOUT operation, L2 extracts the tuples belonging to the required object from database pages at L1's interface. Instead of sending whole data pages to the workstation, L2 sends to OSL only the data which form the required object. These data are transferred to the workstation as sets of tuples. OSL keeps these sets of tuples together in a so-called object-oriented representation at the workstation. To manage the object data in main memory (and control the communication with the disk unit), OSL realizes a so-called object-oriented buffer manager. At OSL's interface, AL manipulates only data in the object-oriented representation. Since all the object data are kept together at the workstation, design tools can access them more efficiently.

Every time a designer at some workstation either checks in an updated object or inserts a new object into the public database on a server, L2 maps the object-oriented representation form of this object onto a page-oriented representation which is first kept in the page/segment buffer controlled by L1. Later, this page/segment representation of the object is saved on stable storage at the server node.

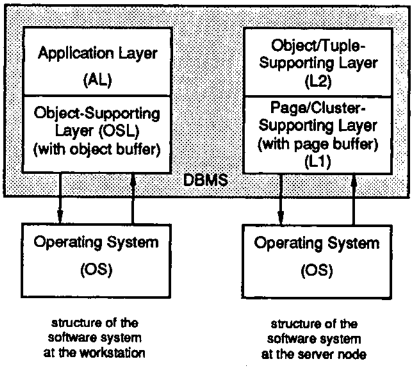

Application Layer (AL)
Object-Supporting Layer (OSL) (with object buffer)
Object/Tuple-Supporting Layer (L2)
Page/Cluster-Supporting Layer (with page buffer) (L1)
DBMS
Operating System (OS)
Operating System (OS)
structure of the software system at the workstation
structure of the software system at the server node

Fig. 1.4: A possible architecture for kernel design database systems

1.3 Main Objectives of the Research Work

Relying on the characteristics of business-related applications and database systems presented in section 1.1 as well as on the description of design applications and design DBS environments presented in section 1.2, we can identify several important differences between business-oriented and design database systems:

- While business-oriented data models provide means for the definition of simple data objects (e.g. flat records or tuples) and support relatively simple database operations either on single records or on tuple sets, the object-oriented data models proposed to support design applications directly support the definition of highly structured data objects which can be manipulated either as single entities or on a record-at-a-time basis.
- While business-related applications are usually characterized by a high locality of access (i.e. different users access common data frequently), the database tends to be partitioned under groups of users in most design applications.Therefore, access conflicts are possibly much more frequent in business-oriented database systems than in non-standard ones (at least when we consider data processing activities at the server node).
- While in business environments transactions (or dialogue steps) are of short duration and execute concurrently following strict isolation, transactions in non-standard environments can take very long and may have to see non-committed results of other running transactions (i.e. they may have to cooperate with each other).
- In business-oriented systems, data are usually processed at the sites where they are stored. This is an efficient strategy for these systems, since in most cases the activity of transferring data to the user site could take longer than the time to execute the transaction. In design applications, though, transactions take much longer than data transfer activities. Therefore, it is cheaper for the system to transfer large amounts of data to the workstation at once than to spawn sub-transactions at the server node every time the public database must be accessed. This strategy creates so-called database hierarchies. That is, each processing node of the system owns and controls a database. Transactions running at one node process data in the node´s own database. If the desired objects are located at another node (i.e. in another database), CHECKOUT operations are started which copy them into the local database.

- While business-oriented DBSs usually realize only one system buffer at a lower system level (e.g. page-oriented level) to better support locality of access, design database systems tend to implement hierarchies of buffers locating them at various system levels. Buffers at higher system levels are used to both reduce the number of operations necessary to map data representation through different system levels as well as explore locality of access at higher levels.

The architectural characteristics of the design environment also induce modifications on already well understood DBS mechanisms such as the concurrency control subsystem or the buffer manager. The former will have to cope with transaction cooperation issues while the latter will have to control data in different representations in main memory. Buffer hierarchies as well as database hierarchies will certainly create new possibilities for recovery in database systems. As a consequence, new recovery techniques may emerge which save and recover data at higher or even multiple levels of abstraction. On the other hand, conventional recovery mechanisms will not be able to cope with processing environments where transaction cooperation is allowed.

The cooperation between server and workstation in non-standard systems also represents a new research topic in database system technology. Depending on how the database system is distributed over the server-workstation architecture, data objects can be exchanged between processing nodes at various levels of abstraction (e.g. at the page level or at the object level). Furthermore, cooperation between server and workstation can be controlled on the basis of different distributed transaction mechanisms (e.g. distributed nested transactions).

While a lot of research effort has been spent in the investigation of appropriate concurrency control techniques for non-standard database systems (especially for design DBSs) (e.g. [KLMP84], [KoKB87], BaRa88], [DüKe88], [GaKi88]), only some investigation concerning the cooperation between server and workstation has been carried out until now (e.g. [DeOb87], [HHMM88]) and even fewer studies have been reported on recovery for non-standard database systems (e.g. [KaWe84], [WeKa84]).

During the research work which resulted in this dissertation, we analyzed database recovery in the design environment. On the basis of the static and dynamic properties of design applications as well as of the architecture of various OODBS prototypes, we investigated database recovery requirements, recovery correctness, and recovery performance in the design environment. In particular, the investigation of database recovery in the design environment is directed at the following goals:

- Identification of the main database recovery requirements posed by the design environment.
- Analysis of the suitability of standard database recovery techniques/mechanisms for design database systems.
- Identification of possible design system properties which can either restrict or even forbid the use of standard database recovery techniques in the design environment.
- Identification of possible design system properties on the basis of which new recovery techniques/mechanisms can be developed.
- The comparative performance study of those database recovery techniques which can be applied to the design environment.

1.4 The Structure of the Dissertation

The remainder ot this dissertation is organized as follows. In chapter 2, we discuss the software architecture of design database systems. We do that by first presenting the architecture of some design database system prototypes which are described in the literature. Then, relying on the architecture of those systems, we derive a reference architecture for database systems in the design environment. In chapter 3, we investigate some well known design processing models which have been proposed in the literature. On the basis of these models, we identify the main characteristics of different transaction management strategies which have been proposed for the design environment. The reference DBS architecture to be derived in chapter 2 as well as the general design processing models to be derived in chapter 3 are used in the following chapters as a basis for our investigation of database recovery in design database systems.

In chapter 4, we analyze database recovery requirements in the design environment. First, a failure model for design database systems is derived which distinguishes the failure modes that should be coped with by the recovery component at the database system level from those which should be treated by other subsystems of the design environment. Then, a set of recovery protocols which can guarantee database reliability is presented and explained. At this point of the dissertation, we distinguish recovery activities in design DBSs which use transaction serializability as a correctness criterion for concurrency from those recovery activities in database systems which allow transactions to cooperate (i.e. exchange non-committed results).

In chapter 5, we investigate the correctness and performance of various existing (and well known) recovery techniques in the scope of the design environment. Besides that, we comment on some newly proposed recovery algorithms which may improve database system performance in that environment. In chapter 6, we report on the performance analysis of a set of recovery algorithms in a design environment which is based on a server-workstation computer system. We simulated the behavior of some recovery techniques (selected from the ones in chapter 5) in a kernel design database system. The simulation model is based on the reference system architecture of chapter 2 and on the general design processing models of chapter 3. Chapter 7 concludes the dissertation. In this chapter, we summarize the work presented in the prior chapters, compare it with the work of others, and discuss some open questions and plans for future work.

Chapter2

A Reference Architecture for Design Database Systems

In the present chapter, the architectures of various non-standard database system prototypes are investigated. Through the study of these architectures, we try to both better understand the static properties of design DBSs (i.e. their structure) and identify common architectural properties of these systems. Relying on the similarities presented by the prototypes studied, we propose a reference model for design DBS architectures at the end of this chapter. Finally, we discuss some of the main aspects involved in the distribution of design DBSs over server-workstation computer systems and propose a distributed architecture for the reference model. In the following chapters, we will rely on the reference model derived here to investigate recovery requirements posed on non-standard database systems by the design environment as well as to carry out a performance analysis of various recovery techniques applied to these systems.

Verhofstad has been probably the first author to propose an architecture for database systems that realizes a buffer hierarchy [Verh79]. His database architecture is a layered one where the lowest layer is represented by the hardware systems and each upper layer represents an abstract machine that implements one ore more abstract data types on the basis of other, less complex abstractions which are, in turn, implemented by lower system layers.

Each software layer of the database architecture consists of a data storage structure, a set of operations for creating, destroying, modifying, and examining the structure, and an algorithm to map the values of the data structure into values of the abstract type being implemented. On the basis of this system architecture, Verhofstad proposes a recovery mechanism for database systems that relies on the recoverability of specific data types. The recoverability of every type is supported by the respective layer which implements it and is to be considered part of the abstract type implementation.

In the architecture proposed by Verhofstad, the storage structures supported by the different system layers form a storage hierarchy where each storage structure stores the same data (or parts of it) in a different representation (i.e. in a different level of abstraction). Since all these storage structures are supposed to be kept in main memory, they constitute a buffer hierarchy.

In [Reut80], Reuter comments on Verhofstad's database system architecture. He argues that this architecture can be used in the study of database systems but not as a reference architecture for existing systems, since no database system at that time realized a buffer hierarchy. Reuter's argument is valid yet. Standard database systems usually realize only one system buffer (e.g. page-oriented buffer). When higher data abstractions (e.g. tuples) are materialized in main memory, they are kept in temporary storage spaces (e.g.

registers, program work spaces) which are not directly controlled by the DBS and, consequently, are not related to buffer management or recovery functions.

Although standard database systems seem to work efficiently without a buffer hierarchy, many non-standard DBS researchers have, at least, felt the necessity of such a mechanism. Since the data abstractions to be realized by higher system layers have become much more complex, many researchers feel that the system would become too inefficient if higher abstractions were to be frequently derived from lower data abstractions.

Performance studies with non-standard DBS prototypes have, at least partly, confirmed this hypothesis. For instance, a test with the DAMOKLES database system [DAM86b] on top of a SUN 3/60 computer has revealed that some 15.000.000 machine instructions must be executed to CHECKOUT an object consisting of 50 subobjects of 50 bytes each. This makes some 300.000 instructions per subobject. Using the same computer, it takes only 20 machine instructions to both access and update a record residing in main memory whose address can be found in a hash table (which is also in main memory).

The CHECKOUT operation is expensive, because it involves a number of complex suboperations. For instance, for each (sub)object **O** being checked out, the system must check if any other of its ancestors has been checked out before. If this is the case, the system should not allow two ancestor objects to be checked out in conflicting access modes. Likewise, the system must investigate all subobjects of **O** in order to check if any of them has already been checked out on behalf of another one of its ancestors. Many of these suboperations are realized through transitive closures which consume very much CPU time. Besides that, the object and the subobjects being checked out may have to be organized in a specific main memory representation in order to be processed at the workstation. CHECKIN operations which either insert new objects or bring updated ones back into the public database may become very expensive, too. They may have to map the main memory representation of complex objects onto database page, update access paths for updated (sub)objects, and release CHECKOUT locks which are kept in stable storage.

In the following, we comment on various database system prototypes which realize buffer hierarchies. We compare their system architectures with the non-standard DBS architecture shown in Figure 1.3. The dynamical properties of the prototypes being presented (i.e. how they implement operations on the database) will be explained through an example of a design transaction.

2.1 Example of a (short) design transaction

To present our transaction example (T), we first need to describe the database on which it operates. It consists of two overlapping (i.e. non-disjoint [BaBu84]) data objects of type *LIST*. This data type is described below.

LIST	::=	*List_Nr Node_List*
List_Nr	::=	integer
Node_List	::=	*Node / Node Node_List*
Node	::=	*N_Nr N_Data N_Next*
N_Nr	::=	integer
N_Date	::=	date
N_Next	::=	integer

The object type *LIST* represents composed objects. Each list instance consists of an attribute which identifies it (*List_Nr*) and a set of (sub)objects of type *Node*. Moreover, the *LIST* type allows the existence of overlapping list instances (as well as of circular list instances). Figure 2.1 depicts the graphical representation of **L1** and **L2** which are both objects of type *LIST* . **L1** and **L2** have two common subobjects, namely the nodes **N1** and **N2.**

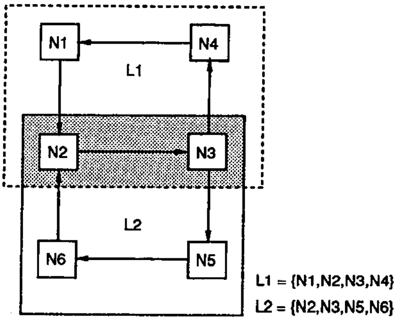


Fig. 2.1: Database state before the execution of T

On the basis of the database described above, we can define the transaction **T**. We suppose the designer interactively works with a graphic editor (GE) at a workstation. On his behalf, GE reads a specific list from the database and displays it graphically on the screen. The designer browses over the list and, sometimes, executes update operations on it. At the end of **T**, GE must ensure that the updated version of the list is brought back into the database.

Transaction **T** can be described as follows:

O1: Begin_Of_Transaction (BOT)

O2: Get (**L1**); Access_Mode (browse, update)

O3: Show_In_Detail (**N2**)

O4: **N2**.N_Data := ′new-value′

O5: Show_In_Detail (**N3**)

O6: Remove **N3** from **all** lists (i.e. delete **N3** in the database)

O7: Insert **N7** into **L1** between (**N2**,**N4**)

O8: End_Of_Transaction (EOT)

Figure 2.2 depicts the state of the database after the execution of **T**.

2.2 The POSTGRES Database System

POSTGRES is a so-called extensible database management system. It is being developed at the University of California at Berkeley. Its data model is an extension of the relational model which can be used to realize some constructs of semantic data models [RoSt87]. POSTGRES supports the definition and operation of abstract data types, the use of user-defined procedures as tuple attributes, and attribute inheritance among object types. With these extensions to the relational model, it is also possible to represent non-disjoint as well as recursive data objects in the database. POSTGRES also supports temporal queries against the database. For this purpose, the system realizes a tuple version mechanism which enables various versions of a same tuple to coexist in the database.

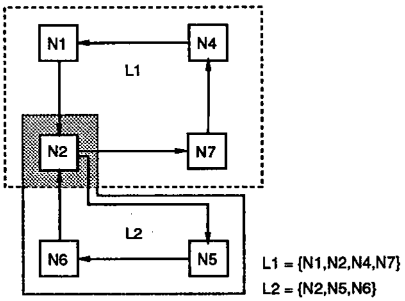


Fig. 2.2: Database state after the execution of T

The system-defined attribute type POSTQUEL can be applied to relational tuples and represents, at the same time, user-defined procedures (which are written either in POSTQUEL [HeSW75] or some procedural programming language) as well as the results of their execution. Moreover, the POSTGRES user can define new database operations (i.e. methods) which are implemented by the system as user-defined procedures. POSTGRES stores the compiled code of POSTQUEL attributes and user-defined methods in the database. In addition, the system realizes a mechanism which controls changes in the definition and in the results of POSTQUEL attributes and methods.

Figure 2.3 illustrates the software architecture of POSTGRES. It consists of two main components: the POSTMASTER and the POSTGRES server [StRo86], [Rowe86]. POSTMASTER manages the database in stable storage, implements the page-oriented system buffer, and realizes transaction management activities. This layer corresponds to the buffer-and-segment manager module of the general system architecture shown in Figure 1.3. One POSTMASTER component will be installed at every computer node of a POSTGRES database system.

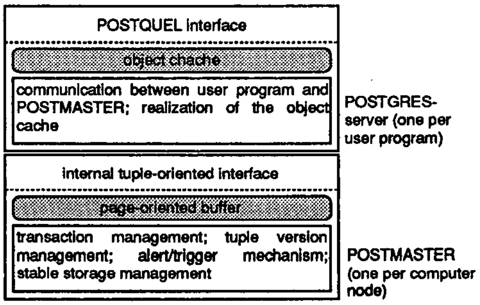

POSTQUEL interface
object chache
communication between user program and POSTMASTER; realization of the object cache
POSTGRES-server (one per user program)
internal tuple-oriented interface
page-oriented buffer
transaction management; tuple version management; alert/trigger mechanism; stable storage management
POSTMASTER (one per computer node)

Fig. 2.3: The software architecture of POSTGRES

Many of the functions realized by POSTMASTER (e.g. lock management, event/trigger mechanism) will be executed by so-called demon processes. These are system processes which run in parallel to user processes. Demon processes can be suspended and restarted later on, depending on the actual system load.

POSTGRES realizes a three-layer storage hierarchy. Old tuple versions are kept on optical disks while current data are stored on magnetic disks. The third layer of the storage hierarchy is represented by the main memory of the system, parts of which should also be stable.

The POSTGRES server (which is also called run-time system) forms the higher layer of the POSTGRES two-layer architecture and uses the POSTMASTER as a backend. This layer corresponds to the four upper layers of the general architecture of Figure 1.3. There exists one POSTGRES server associated with every active user program in the system. The servers are realized as independent processes which control the execution of database operations on behalf of user programs. Although each server supports the execution of only one user program, it can process various database operations in parallel [StRo86].

Each running POSTGRES server maintains a so-called object cache [Rowe86]. The object cache is an object-oriented buffer which relies on the portal concept [StRo84]. In the cache, the server stores temporary transaction results (as sets of tuples) as well as frequently accessed results of POSTQUEL attributes and user-defined methods (as sets of tuples or multirelations). The object cache has not been completely realized in the actual POSTGRES implementation. Update operations are not directly executed in the cache. Tuples are first updated on pages (at the page-oriented buffer) and then brought into the cache again.

Concerning the classification of recovery strategies in [HäRe83], the recovery mechanism of POSTGRES can be defined as (non-atomic, steal, force, toc). Transaction results are forced to the database on disk at transaction commit.

Since the system maintains demon processes which are continually removing tuple versions created by transactions which have been backed out, almost no recovery activity is necessary to either abort transactions or restore the database state after system crashes. Moreover, POSTGRES will support recovery from media failure on the basis of mirrored disks.

One of the possible ways of modeling our database example in POSTGRES is presented below. The description of each list node defined in the database is kept in the Nodes relation. Each node-next-node relationship is kept as a tuple in the Next_Node relation. Each existing List is described by a single tuple in the Lists relation. Tuples of this relation consist of two attributes. One attribute of type char (i.e. string of characters) and another one of type POSTQUEL. The latter represents a user-defined procedure and its current result at the same time. The list_procedure returns either a list of nodes or the empty set.

```
Create Nodes (N_Nr=char[2], N_Data=char[])
   Key (N_Nr)
Create Next_Node (L_Nr=char[2], N_Nr=char[2], N_Next=char[2])
   Key (L_Nr, N_Nr)
Create Lists (L_Nr=char[2], List=list_procedure)
   Key (L_Nr)
```

```
Define type list_procedure is
        Retrieve (Chain = n.N_Nr, n.N_Data, x.N_Next)
            from n in Nodes, x in Next_Node
                where x.L_Nr = $.L_Nr and x.N_Nr = n.N_Nr.
                (note: $ identifies the current list in Lists)
```

The POSTQUEL attribute List of the Lists relation represents for each list tuple the set of nodes that compose it. POSTGRES keeps the compiled code of list_procedure in the database. Moreover, by means of demon processes the system executes the procedure for each tuple in Lists and stores the resulting tuple sets as materialized views in the database. When a user program accesses the relation Lists, the POSTGRES server associated to it pre-fetches the List subrelations and stores them in the object cache.

By executing the operation O2 of **T**, the server brings the list **L1** into the object cache. Since the operations O3 and O5 only read data, they can be directly executed in the object cache. The operations O4, O6, and O7, on the other hand, must be processed in the page buffer. Their results are then brought into the cache.

The operation O6 (i.e. the deletion of node **N3** in the database) causes the attribute List in all tuples of the Lists relation to be marked invalid. On the basis of a deferred-update mechanism, these attributes will be once again calculated either the next time they are accessed or at a moment when the system becomes idle. By the end of **T**, all data it updates must have been copied to stable storage.

2.3 The Darmstadt Data Base System (DASDBS)

The DASDBS is a non-standard database system prototype formerly being designed and implemented at Darmstadt University. It actually represents a family of database systems where a database system kernel supports various, application-specific system modules.

The DASDBS's kernel provides an extended relational data model at its interface, namely the NF^2 data model (where NF^2 stands for Non-First Normal Form) [ScWe86]. This model supports the definition of relation-valued tuple attributes. Thus, subobjects can be expressed as subrelations in the database. For the manipulation of the resulting hierarchies of relations, the model provides a set of recursive relational operations (i.e. the NF^2 relational algebra). The NF^2 data model supports only disjoint complex objects, that is, objects which have no common (sub)objects.

DASDBS has a three-layer software architecture as it is shown in Figure 2.4. The two lower system layers (i.e. SMM and CRM) build the kernel of DASDBS The uppermost layer (i.e. AOM) supports application-specific data models. Compared with the layers of the general architecture in Figure 1.3, SMM corresponds to the buffer-and-segment layer while CRM integrates the functions of the record manager, complex object manager, and query processor. Finally, AOM realizes the application layer.

The stable memory manager (SMM) controls the data organization on disk and realizes the data transfer between this device and the system on the basis of the block concept [DePS86]. That is, sets of pages which are stored at neighboring addresses on disk can be transferred to/from disk in only one I/O-operation. The block concept is also known as "chained-I/O" SMM manages a page/segment-oriented system buffer and supports a set-oriented page interface.

Relying on SMM's interface, the complex record manager (CRM) realizes the NF^2 data model. NF^2 tuples are actually represented in the database as sets of flat tuples together with CRM information about their logical connections. Since NF^2 objects are always disjoint, complex objects can be represented as object trees in the database. CRM takes advantage of the relative simplicity of the data model and stores object trees as a whole on adjacent database pages. These pages form the so-called object clusters which then can be read from disk or written to it in only a few I/O-operations. In this way, the time to read/write complete objects is reduced in the system.

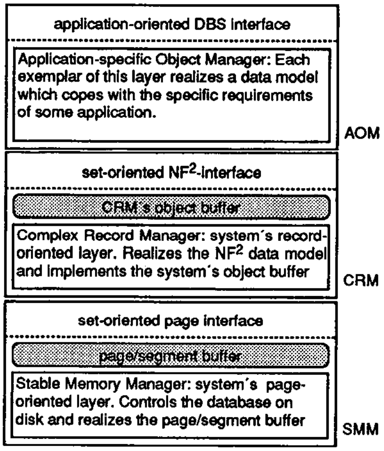

application-oriented DBS interface
Application-specific Object Manager: Each exemplar of this layer realizes a data model which copes with the specific requirements of some application.
AOM
set-oriented NF^2-interface
CRM's object buffer
Complex Record Manager: system's record-oriented layer. Realizes the NF^2 data model and implements the system's object buffer
CRM
set-oriented page interface
page/segment buffer
Stable Memory Manager: system's page-oriented layer. Controls the database on disk and realizes the page/segment buffer
SMM

Fig. 2.4: The software architecture of DASDBS

CRM realizes a set-oriented NF^2 interface, that is, higher system layers can define and manipulate sets of NF^2 tuples at CRM's interface. To reduce the impedance mismatch between the CRM and AOM programming environments, DASDBS realizes an object buffer as part of CRM's interface [Paul87]. The object buffer can store NF^2 tuple sets which then can be accessed in a tuple-at-a-time manner by the application-specific system modules. There exists one object buffer for each user transaction in the system, that is, these data repositories are not shared among transactions. Besides retrieve and browsing operations, CRM also implements some update operations which directly manipulate data in the object buffer. One of the main goals of DASDBS is to implement the whole set of CRM operations on the basis of the object buffer.

By a possible distribution of DASDBS over a server-workstation architecture, the object buffer could support the communication between the public database system at the server node(s) and the local database systems at the workstations [DePS86].

The application-specific object manager (AOM) realizes the application's view of the data and supports user transactions. AOM layers rely on the DASDBS kernel interface (i.e. CRM's interface) to realize application-specific data models (e.g. office-filing model [PSSW87]). These models should capture the specific semantics of the applications they support. Application-oriented data types and operations are realized by AOM in the form of abstract data types on the basis of NF^2 relations. AOM also implements application-oriented access paths which are not realized by lower system layers. On the basis of AOM, DASDBS can support behavioral object orientation.

DASDBS implements a multi-level transaction management strategy [Wei87a]. Multi-level transactions represent a special case of nested transactions [Moss81] and rely on the open-transaction concept which is described in [Trai83] and was implemented in System R

[Gray81]. In a multi-level transaction environment, each level of the nested transaction hierarchy is associated with a specific system layer. Therefore, each of these system layers realizes a transaction manager which controls the execution of the (sub)transactions that are associated with the layer.

In DASDBS, user transactions are controlled by AOM. These transactions (which will be called AOM-transactions from now on) are composed by one ore more operations of the AOM interface. AOM operations are implemented by AOM on the basis of CRM operations. The CRM transaction manager considers each set of its operations that implements a specific AOM operation to constitute a CRM transaction. Each CRM operation is, in turn, partly realized by a set of SMM operations. The SMM transaction manager considers sets of SMM operations that support CRM operations to be SMM transactions.

Multi-level transaction management can improve transaction parallelism by avoiding so-called pseudo conflicts. These are access conflicts which can happen in lower system layers, although they do not exist in (and could be avoided by) higher layers. A typical example of pseudo conflict is the one of two concurrent transactions (e.g. transactions **ta** and **tb**) which update different data records that are stored on the same data page. If transaction management is realized only at the page level and **ta** accesses the page before **tb**, the latter will have to be blocked (or even backed out) during the whole execution of the former (supposing strict two-phase locking). In a multi-level transaction environment (at the record level and page level, for instance), **tb** will be blocked only for the time **ta** is either reading a value from or writing a value to the page, because the transaction manager at the system´s record level knows that the two transactions are actually processing different data objects.

To cope with multi-level concurrency control, the recovery mechanism of DASDBS must also be distributed over different system layers [Wei87b]. SMM saves page updates at the commit phase of every SMM-transaction. In this way, this layer guarantees both partial and global redo recovery in case of system crash or media failure (see [HäRe83] for explanations on this terminology). Partial and global undo recovery are supported by the recovery managers of higher system layers (i.e. CRM and AOM) which save undo information about their respective transactions.

When defining our database example in DASDBS, we must have in mind that overlapping data objects (e.g. **L1** and **L2**) are not supported by this system. One possible way to define the list objects using the NF^2 data model is presented below. Tuples in Lists represent existing objects of type *List*. For each tuple in Lists, L_Nodes contains all node-next-node relationships related to the respective list object. Tuples in Nodes describe all existing objects of type *Node*. For each tuple in Nodes, N_Owners contains all node-list relationships related to the respective node object.

```
Crate Lists    (  L_Nr=char[2],
                  L_Nodes=(N1_Nr=char[2], N2_Nr=char[2] )
               )
Create Nodes (    N_Nr=char[2], N_Data=char[],
                  N_Owners=(NL_Nr=char[2])
               )
```

(note: both L_Nodes and N_Owners are relation-valued tuple attributes and represent subobjects of Lists tuples and Nodes tuples, respectively)

Since the kernel of DASDBS does not realize logical access paths, the relation Nodes will be used by AOM as an index. On the basis of this relation, AOM can also control the data redundancy introduced in the system. Each tuple in Nodes describes a 1:N relationship between a list node and all the lists to which it belongs.

To read the ring list **L1** into the private object buffer **OB**, AOM executes O2 (which is expressed in a SQL-like query language) at CRM´s interface:

O2: Select into the Object_Buffer **OB** Lists.all
    where Lists.L_Nr = ´**L1**´

The complex record manager, in turn, starts a CRM-transaction which executes the AOM operation. It identifies in which cluster the list **L1** is stored and reads it via the SMM interface. The stable memory manager starts an SMM-transaction which identifies the cluster address on disk and reads it into the page/segment buffer in one I/O-operation. The CRM-transaction, then, reads **L1** data into **OB**.

By analysing the L_Nodes subrelation of **L1** in **OB**, AOM can decide which tuples of the Nodes relation should be read from disk. After doing that, AOM starts another operation to read the required Nodes tuples. To execute this operation, CRM starts a new internal transaction which reads one ore more page clusters via SMM. These read operations, in turn, cause SMM to start new SMM-transactions.

The browsing operations O3 and O5 as well as the simple update operation O4 and the insertion of node **N7** (by O7) can be directly executed in **OB**. Although O6 can be partly executed in **OB**, this operation must be complemented as **OB**´s updates are mapped onto database pages at **T** commit, since O6 also implies the change of other data which are not stored in OB (e.g. the list **L2**). At commit time (O8), CRM must transfer all data which have been updated in **OB** to SMM. This procedure, in turn, involves the execution of new SMM-transactions.

2.4 The R^2D^2 Interface of the AIM System

The Relational Robotics Database System with Extensible Datatypes (R^2D^2) has been developed at the University of Karlsruhe in cooperation with the IBM Heidelberg Scientific Center. This system realizes an abstract data type (ADT) facility on the basis of the NF^2 data model. R^2D^2 supports the execution of ADT operations which are called from inside of Pascal programs [KeWa87]. To accelerate data processing activities at the application level, R^2D^2 transforms the NF^2 representation of database objects into some equivalent Pascal representation of them. Therefore, application programs can directly process dabase objects in the user address space.

The R^2D^2 system is being built on top of the AIM-P database system prototype [KuDG87] to be used as a specialized object-oriented database system supporting engineering applications. The Advanced Information Management Prototype (AIM-P) has been developed at the IBM Heidelberg Scientific Center. It is a non-standard DBS which realizes the NF^2 data model. This system presents a layered system architecture and was designed to support server-workstation environments.

The software architecture of R^2D^2 is shown in Figure 2.5. Its two lower layers represent the AIM-P system. AIM-P´s architecture resembles the DASDBS kernel system. Its lower software layer manages the database on disk and realizes the page-oriented buffer. The next higher layer incorporates a set of system modules which together realize the NF^2 interface.

In [KüDG87], an object buffer for AIM-P´s higher software layer is described. An object buffer instance at the server node stores NF^2 data objects during their construction as well as during their reintegration into the database after user processing activities. The user at the workstation processes objects in another object buffer instance which is realized at that processing node. The object buffer concept has not been realized in the actual AIM-P version, though.

AIM-P realizes transaction management at the tuple level which is represented in Figure 2.5 as part of the NF^2 system layer. This layer supports a tuple-oriented locking mechanism and a recovery mechanism that relies on the multi differential-file concept described in [SeLo76] and [HaKK81]. A complete description of this mechanism can be found in [KHED89].

The third and highest layer realizes the user interface of the system. Currently, AIM-P supports two distinct user interfaces: the application programming interface (API) and the R^2D^2 interface. API implements an embedded NF^2 interface. It embeds both DDL and DML statements of the NF^2-oriented database language HDBL into the programming language Pascal [ErWa86], [ErWa87]. The R^2D^2 interface realizes an extended HDBL which supports the definition and operation of ADTs

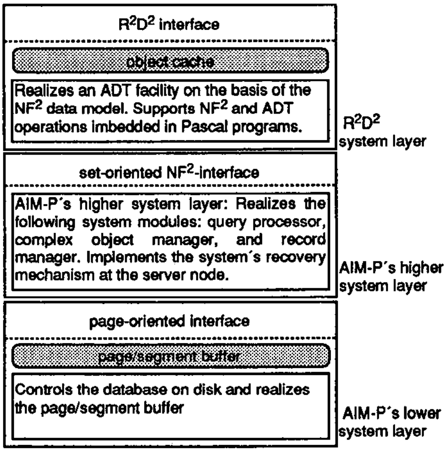

R^2D^2 interface
object cache
Realizes an ADT facility on the basis of the NF^2 data model. Supports NF^2 and ADT operations imbedded in Pascal programs.
R^2D^2 system layer
set-oriented NF^2-interface
AIM-P's higher system layer: Realizes the following system modules: query processor, complex object manager, and record manager. Implements the system's recovery mechanism at the server node.
AIM-P's higher system layer
page-oriented interface
page/segment buffer
Controls the database on disk and realizes the page/segment buffer
AIM-P's lower system layer

Fig. 2.5: The software architecture of R^2D^2

The whole system realizes a so-called database hierarchy. While the AIM-P modules realize a kernel DBS which controls the public database at the server computing node, the R^2D^2 layer manages the local database at the workstation. There exists one local database for every active application program in the system. Figure 2.6 which has been taken form [KeWa88] depicts R^2D^2's database hierarchy.

Besides controlling the data transfer between databases as well as the data processing activities at the workstation, R^2D^2 also realizes a locking scheme for CHECKOUT and CHECKIN operations and implements a recovery mechanism for nested transactions running at that processing node [DüKe88], [Ries89].

Engineering transactions execute in R^2D^2 according to the following processing scheme. NF^2 data objects are checked out of the public database on the server through the execution of ADT operations. AIM-P builds the NF^2 representation of the objects being checked out and stores them in a file (i.e. the result table) on disk, after they have been locked in the public database. When the whole set of NF^2 objects being checked out have already been written to the result table, this file is transferred to the workstation. At the workstation, the result table is stored on disk and represents a part of the user´s private database (i.e. the local database). The second part of the private database is represented by the object cache. The cache is an object-oriented system buffer located in the address space of the application programm in main memory. Through the execution of fetch and pre-fetch operations which are started by the application program, R^2D^2 reads NF^2 objects in the local database, translates them into Pascal data structures, and brings them into the object cache. The application program, then, can process the resulting Pascal data structures in the cache.

Database objects in Pascal representation are mapped back to their NF^2 representation in the local database either if this operation is explicitly invoked by the application program or when the object cache becomes full (release operation). Depending on the lock mode being applied to the CHECKOUT operation, updated NF^2 objects can be propagated to the public database on the server either during transaction execution or at transaction commit [DüKe88]. As with CHECKOUT operations, the system uses the result table to transfer updated data objects from the local DB at the workstation into the public DB at the server during CHECKIN.

Our database example can be defined as an abstract data type at the R^2D^2 interface. The control of data redundancy cannot be integrated into the definition of the ADT, though, since the ADT facility relies upon the NF^2 data model. By defining an ADT, the user must specify both its data structure and the operations it must support. The user must declare the NF^2 and the Pascal representations of each ADT. ADT operations may invoke operations on other ADTs. A possible alternative of modeling the database example in R^2D^2 is presented below.

```
Create ADT Lists
  {
     Structure is    (
                          L_Nr=char[2],
                          L_Nodes=(L_NNr=char[2], L_NNext=char[2])
                     )
     Operations are:      insert_node_in_list (L_Nr, L_NNr, prior, next); ... ;
  }
Create ADT Nodes
  {
     Structure is    (
                          N_Nr=char[2], N_Data=char[],
                          N_Owners=(N_LNr=char[2])
                     )
     Operations are:      insert_owner_in_node (N_Nr, N_LNr); ... ;
  }
```

Operation O2 of T is equivalent to a CHECKOUT in R^2D^2. By the execution of O2, the NF^2 representation of the list object **L1** is first built in the server´s main memory and stored in the result table. After that, the result table is transferred to the workstation where it is integrated into the local database. If O2 is started by the application program and the pre-fetch option is set, R^2D^2 immediately transforms the NF^2 representation of **L1** into its Pascal representation and stores it in the application program´s private object cache. After the completion of O2, all other operations of **T** but O6 can be executed by the application program directly in the cache. Since O6 modifies data objects which are neither in the object cache nor in the local database (e.g. **L2**), this operation must also be executed in the public database. From what has been published in the literature, it became not clear whether O6 will be executed on the server node even before **T** terminates or only at **T** commit.

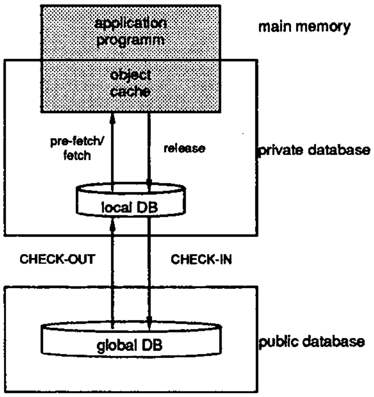


Fig. 2.6: R^2D^2's database hierarchy

At the end of **T** at the latest, the system maps the updates back onto the NF^2 format, stores them in the result table and sends this file to the server node. There, AIM-P propagates the results to the database by storing updated NF^2 tuples onto pages of the public database.

2.5 The PRIMA Database System Prototype

The PRIMA project was developed at the University of Kaiserslautern. Its main objective is the construction of a DBS prototype to support engineering applications [HMMS87]. PRIMA is being realized as a kernel database system supporting the Molecule Atome Data Model (MAD) [Mits87]. On the basis of this general purpose object-oriented data model, application-oriented system layers should be realized to support application-specific requirements.

MAD supports structural object orientation. It provides means for the definition as well as the manipulation of data objects as structured sets of elementary building blocks [HüMi88]. The model supports the dynamic definition of object types as well as the derivation of new types from the composition of existing ones. MAD permits the creation of non-disjoint object instances and supports N:M object relationships. The model provides a descriptive SQL-like query language (MQL) which provides set-oriented object processing facilities.

Figure 2.7 illustrates PRIMA's software architecture. It consists of the kernel database system and the application-specific layer which realizes the application model interface. Conceptually, the kernel system implements three levels of abstraction which are built on top of each other forming a hierarchical architecture. The software layers that implement such levels of abstraction, namely the data, access, and storage systems, map MAD objects onto data blocks kept on external storage.

Compared to the system layers of the general architecture presented in Figure 1.3, the data system combines the features of both the query processor and the complex object manager while the access system is equivalent to the record manager, and the storage system realizes the functions of the buffer-and-segment manager.

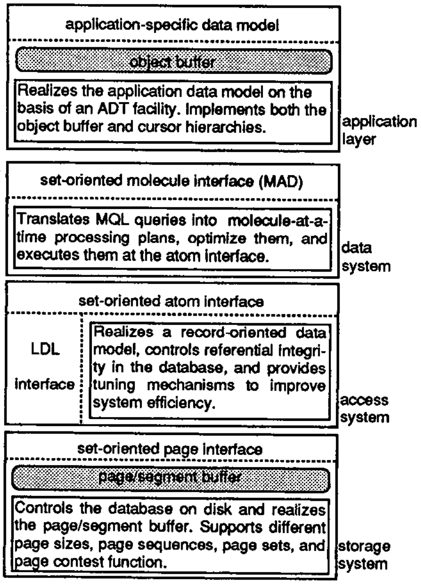

application-specific data model
object buffer
Realizes the application data model on the basis of an ADT facility. Implements both the object buffer and cursor hierarchies.
application layer
set-oriented molecule interface (MAD)
Translates MQL queries into molecule-at-a-time processing plans, optimize them, and executes them at the atom interface.
data system
set-oriented atom interface
LDL interface
Realizes a record-oriented data model, controls referential integrity in the database, and provides tuning mechanisms to improve system efficiency.
access system
set-oriented page interface
page/segment buffer
Controls the database on disk and realizes the page/segment buffer. Supports different page sizes, page sequences, page sets, and page contest function.
storage system

Fig. 2.7: PRIMA´s software architecture

The lowest layer, namely the storage system, realizes a set-oriented page interface and implements the system´s page/segment-oriented buffer. This layer supports five different page sizes: 0.5, 1, 2, 4, or 8 Kbytes.

The storage system interface offers its users (e.g. the access system) three different page set concepts [HMMS87]. The **page sequence** concept resembles the page cluster concept. It treats an arbitrary number of pages as a whole. Page sequences are supported by a cluster mechanism of the underlying file system. It guarantees that pages belonging to the same page sequence are stored at neighboring addresses on disk and are transferred to and from disk in a few I/O-operations (i.e. chained-I/O).

The second concept implemented by the storage system is the one of a **page set**. It may be used by higher system layers to FIX/UNFIX a number of pages (or page sequences) in the page/segment buffer at the same time. Finally, the storage system supports the **page contest** concept on the basis of which higher layers can implement data redundancy in the database (see below). By receiving a list of some/all pages which store copies of a specific record, the storage system calculates the cost of accessing each of the pages and decides, on the basis of a cost analysis, which copy of the record should be provided to the user.

Based on the storage system interface, the access system realizes a set-oriented atom interface which supports the retrieval and update of single atoms (i.e. typed data records) as well as of atom sets. On the basis of so-called reference atom attributes, this system guarantees referential integrity in the database. This type of attribute is used to represent relationships between atoms. PRIMA supports back references automatically. Relationship information is kept replicated in all atoms involved. If the user updates this information in one atom, the access system actualizes the other atoms which participate of the relationship.

This system also implements tuning mechanisms which enable the physical organization of the database on disk to be tailored to the special needs of the application. The database administrator has access to these mechanisms through the Load Definition Language (LDL). Through the tuning mechanisms the user can, for instance, define database views which should be materialized on disk or demand that atom instances of specific types be kept duplicated (or replicated) in the database so that they can be accessed together with their respective ancestor objects more efficiently. The tuning mechanisms also support the definition of application-specific access paths and sort orders. To reduce response time the access system realizes a deferred update strategy to update atom copies. New references to an updated atom must always yield its most up-to-date copy, though.

The data system maps the molecule-oriented interface onto the atom-oriented interface of the access system. It consists of two main components: the query processor and the molecule manager. The former checks user queries for syntatic and semantic correctness, tries to optimize their execution plan, and translates them into data processing plans which rely upon the one-molecule-at-time interface of the molecule manager. This system component consists of two parts. The molecule-type-specific optimization tries to create an optimal processing plan for the query on the basis of existing access paths, sort orders, and data replication. The second part of the molecule manager, namely the molecule processing, executes the processing plans which are developed by the optimizer.

While PRIMA´s kernel system supports structural object orientation by means of the MAD data model, the application-oriented layer (AL) supports behavioral object orientation through the realization of application-specific abstract data types which, in turn, rely on the underlying MAD model [HüMi88]. Application data objects are realized by AL through objects defined at the MAD level (i.e. at the kernel´s interface). The definition of application-oriented algorithms which describe the behavior of application objects is implemented by AL as MQL queries. The encapsulation of application objects and the operations associated with them results in a kind of ADT facility which is provided to the application at the AL interface.

The software architecture of the application layer was presented in [HüMi88]. To support the definition and efficient operation of ADTs, AL realizes the concepts of object buffer and cursor hierarchy. MQL query results (i.e. MAD molecules and atoms) are stored in the object buffer in a special main memory representation. In this special format, data objects can be processed by further ADT operations more efficiently. Such operations navigate through highly structured molecules in the object buffer by means of hierarchies of cursors. Each cursor of a cursor hierarchy can visit only the (sub)objects located at a specific level of the corresponding object hierarchy. When a cursor is moved by an ADT operation from one (sub)object onto the other, AL automatically updates the positions of all its subordinate cursors at their respective levels.

Since the MAD data model supports the definition of non-disjoint objects, we can model our database example as a single molecule type. First, the atom types Node and List are defined using MQL statements. Then, the materialization of the list L1 in the database (on the basis of the atom definitions) is achieved through an LDL statement.

```
Define ATOM Type Node
    ( N_S       : identifier, /* surrogate: created and maintained by the system */
      N_Nr      : char[2],
      N_Data    : char[],
      N_Next    : set_of ( ref_to ( Node.N_Prior)),
        N_Prior    :  set_of ( ref_to ( Node.N_Next)),
        N_Lists    :  set_of ( ref_to ( List.L_Nodes))
    )
    keys_are (N_Nr)
Define ATOM Type List
    ( L_S          :  identifier, /* surrogate: created and maintained by the system */
      L_Nr         :    char[2],
      L_Nodes      :  set_of ( ref_to ( Node.N_Lists))
    )
    keys_are (L_Nr)
Define static molecule type List_L1
    From List
        Where L_Nr = 'L1'
```

At the interface of the application layer, the operations of **T** can be defined as parameterized ADT operations which access molecules of type List_L1 and the atoms related to them. As an example, we define O2 below.

```
Interface:   O2 ( List_Name : char[2])
Body:        Select (into the object buffer) all
                 From List_Name
```

After list **L1** has been brought into the object buffer, all following operations of **T** but O6 can be completely executed there. Since O6 manipulates data which are not stored in the object buffer (e.g. list **L2**), this operation must be complemented by the kernel system (possibly at T commit).

2.6 A Reference Model for Database Systems with Buffer Hierarchies

In the previous sections, we have discussed the software architecture of various non-standard database system prototypes which either suggest to or actually have buffer hierarchies to support complex object processing. Buffer hierarchies have also been implemented by systems which try to integrate existing object-oriented programming environments with database environments. The Object eXchange Service module (OXS) described in [PaJF89], for instance, supports the translation of the main memory representation of Lisp data objects into a stable storage representation which can be managed by the Flavors database system. OXS implements a buffer hierarchy in much the same way R^2D^2 does.

From the systems discussed above, we can conclude that a reference model for non-standard DBSs which implement buffer hierarchies should basically consist of a three-layer software architecture as the one shown in Figure 2.8. The lowest layer (L1) would manage the database in stable storage and realize a set-oriented page interface which would abstract from details concerning data organization on external storage devices. To provide efficient access to database pages, L1 should be able to transfer sets of pages as a whole between disk and main memory and support a page/segment-oriented buffer to reduce IO activity.

The second layer (L2) would rely on the page interface implemented by L1 to realize a structurally object-oriented data model. In both DASDBS and R^2D^2, this data model is the

NF2 model. In PRIMA, on the other hand, it is represented by the MAD model. Both models provide structural but not behavioral object orientation, since they provide means for defining and manipulating highly structured data objects as a whole but do not support features as encapsulation, user-defined types or classes, or inheritance. L2 should be designed to support two complementary functions. First, it should be able to construct the main memory representation of complex objects out of their representation on database pages and map them back onto pages. On the basis of the main memory representation of objects, L2 would implement the operations it provides at its interface.

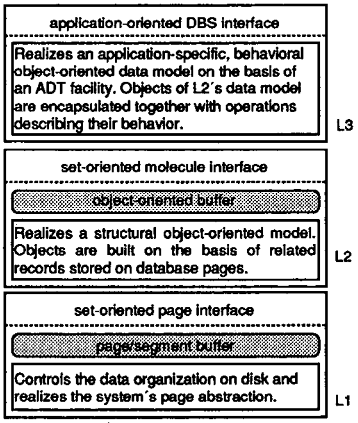

application-oriented DBS interface
Realizes an application-specific, behavioral object-oriented data model on the basis of an ADT facility. Objects of L2's data model are encapsulated together with operations describing their behavior.
L3
set-oriented molecule interface
object-oriented buffer
Realizes a structural object-oriented model. Objects are built on the basis of related records stored on database pages.
L2
set-oriented page interface
page/segment buffer
Controls the data organization on disk and realizes the system's page abstraction.
L1

Fig. 2.8: A reference model for DBSs which realize buffer hierarchies

The uppermost system layer (L3) would realize the application-oriented data model. Such a data model should be able to capture and represent the specific semantics of the application. Relying on the structurally object-oriented model implemented by L2, L3 would support behavioral object orientation. As in R^2D^2 and PRIMA, this could be done through the implementation of an ADT facility which would support user-defined operations on data objects of the L2 interface.

When comparing our reference model with the general non-standard DBS architecture of Figure 1.3, it becomes clear that L3 realizes the application layer, L2 implements the functions of the query processor, complex object manager, and record manager, and L1 implements the buffer-and-segment manager. We decided to allocate the complex object and record managers in the same layer (i.e. L2), because we feel the functions of these two software modules can be integrated to improve system performance.

An important decision concerning the buffer hierarchy of the reference model is the one related to the location of the object buffer in the system architecture. While some systems see the object-oriented buffer as an extension of L2 (e.g. POSTGRES and DASDBS), there are others which consider it to be an integral part of L3 (e.g. R^2D^2 and PRIMA).

Actually, PRIMA's object buffer also represents an extension of the kernel interface, since it is used to represent molecules of the MAD data model. On the other hand, it is controlled by the application layer. In [HHMM88], where considerations about the distribution of PRIMA over a server-workstation architecture are presented, the object buffer together with the cursor hierarchy concept are implemented at the workstation by yet another software module (i.e. the object-supporting layer) which supports the application-oriented layer, controls the local database at the workstation, and realizes the communication between the local data system and the kernel system at the server node (see Figure1.4). R^2D^2's object cache, on the other hand, is really part of the application-oriented layer. The Pascal representation of database objects is not even known at the NF^2 interface. If the object buffer of AIM-P had been implemented, R^2D^2 would actually have realized a three-level buffer hierarchy.

We decided to represent the object buffer in our reference model as an extension to the interface of L2. As part of the structurally object-oriented interface, the object buffer can also be used to improve data communication between system modules in case the reference model is distributed over a server-workstation computer system.

2.7 Distribution Aspects of Non-Standard Database Systems

As already mentioned, database systems supporting the design environment will probably be distributed over server-workstation computer configurations. In this section, we discuss some important aspects concerning the distribution of database software in the design environment.

The DBS software can be either completely replicated in every processing node of the underlying computer system or distributed according to the functions which are executed at each node. While it is relatively easy to identify internal interfaces in kernel architectures on the basis of which the distribution of the database software could take place, the task of distributing the software modules of tailored design DBSs can become a tricky one.

If we imagine a server-workstation system where the server processing node does not directly support application programs and a kernel database system similar to our reference model, it would possibly be a good choice to place the DBS kernel at the server site and implement the application-oriented system layer at the workstation. In this case, the distribution of the DBS software would follow a functionality criterion.

Since objects are kept in the local database of the workstation during the design transaction, another DBS layer must be provided at this node to control the local database and to communicate with the kernel system at the server node. Some real systems solve this problem by installing a copy of the kernel system at the workstation ([DeOb87], [DAM88b]). Application objects, then, are mapped through the whole DBS architecture every time they are read from disk or written to it during the design transaction. A possibly more efficient solution could be to implement a simpler software module on the bottom of the application layer which can directly save complex objects on disk whithout having to change their data representation. That is, objects are written to disk in their main memory format. The PRIMA prototype follows this strategy by the implementation of the workstation data system [HHMM88].

2.7.1 Abstraction Levels for Cooperation

The cooperation between server and workstation can be explained as follows. The application programs run at the workstation and are supported by the application layer. This layer relies on some DBS software (e.g. DBS kernel or another type of object-supporting software layer) to access the necessary objects in the local database. Every time the application layer requests an (complex) object that is not stored in the local database at the workstation, a CHECKOUT operation must be started. Before the design transaction terminates, updated objects must be copied into the public database at the server node. This is done through CHECKIN operations.

Server and workstation can exchange data at different abstraction levels of the DBS architecture. In [DeOb87], the alternatives for communication have been subdivided into single-level cooperation and multi-level cooperation. As the name suggests, single-level cooperation implies that the whole server-workstation cooperation is realized at only one abstraction level of the database architecture. Multi-level cooperation, on the other hand, implies that data are transferred at various levels of abstraction.

The cooperation between server and workstation in DAMOKLES, for instance, is completely implemented at the complex object layer of the system [DAM88b]. That is, for CHECKOUT the workstation sends its request as a complex object request (i.e. EODM statement [DAM86a]) and the server sends the workstation the required object in its main memory representation. For CHECKIN, the workstation sends the server updated EODM-objects. PRIMA, on the other hand, realizes a multi-level cooperation strategy. For CHECKOUT, requests are transmitted by the workstation as complex object queries and CHECKOUT data are transferred by the server in the form of sets of complex objects (i.e. sets of molecules) [HHMM88]. For CHECKIN, the workstation sends the server only sets of updated records (i.e. sets of atoms). In [DeOb87], another multi-level cooperation strategy has been proposed. For CHECKOUT, requests are sent as complex object queries but CHECKOUT data are transferred in form of sets of pages. For CHECKIN, the server receives a set of modified pages as well as some meta information. The meta information is then used to update higher-level data abstractions (e.g. index entries for tuples). This cooperation strategy is suitable for DBS implementations where a copy of the kernel system is installed at the workstation. In this case, both server and workstation present all layers of the DBS architecture.

2.7.2 Supported Transaction Classes

To realize integrated information systems, non-standard DBSs must be capable of executing both, design and conventional transactions. In the case of a kernel design DBS, we can imagine that the software at the workstation would be developed to support only design applications, while the DBS kernel at the server node would execute CHECKOUT/IN operations as well as conventional, short transactions (e.g. relational queries and short update operations).

2.7.3 Transaction Management Strategies

Because of the long duration of design transactions, it would be unacceptable to control transaction parallelism in design systems on the basis of optimistic algorithms, since the costs of design transaction backout would be very high. Therefore, most systems plan to realize some locking method to control parallelism among design transactions.

Usually, standard database systems realize the concurrency control mechanism at a specific layer of the DBS architecture. For lock-oriented synchronization techniques, concurrency control is often implemented at the page level of the system. In this way, the overhead represented by lock management activities is kept low because the granularity of locks is large (in comparison to record locking, for instance) and, consequently, fewer locks are necessary. On the other hand, page locking strategies usually force the system to lock more data than it actually needs (e.g. by requesting one record, the whole page where this record is kept on will be locked for the transaction). As a consequence, parallelism can be reduced.

In design systems which exchange data between server and workstation at the page level, the locking mechanism must be realized at the page level, anyhow. Page locks are acquired during CHECKOUT operations and released only when the objects are checked back into the public database.

Design database systems which exchange data at higher levels of abstraction (e.g. record level or complex object level) can implement multi-level transaction managers ([Lync83], [Wei87b]) to improve parallelism at the server node. In this case, locks on pages are kept only for the time needed to either build the object representation by CHECKOUT or map updated objects back onto database pages by CHECKIN. During the time objects are processed at the workstation, only record locks or object locks are kept at the server system. It is expected, that multi-level transaction management can significantly improve parallelism in database systems [Wei87a].

2.7.4 Controlling the Cooperation Between Server and Workstation

When designing a distributed design database system, it must be decided how the cooperation between server and workstation will be controlled. This cooperation is materialized by the execution of CHECKOUT and CHECKIN operations as well as by remote queries which are started at the workstation and processed by the public system at the server node.

There are, at least, two ways of controlling the cooperation between server and workstation [HHMM88]. In the first alternative, the public system is not really aware of the existence of the design transaction at the workstation. CHECKOUT and CHECKIN operations as well as remote queries are started at the workstation as independent, short transactions (i.e. so-called recovery transactions) which spawn sub-transactions at the server node. The execution of these distributed transactions is synchronized by means of a two-phase commit protocol. Therefore, at the end of a remote operation at the server, all locks it holds in the public database are released (supposing concurrency control at the server is realized by locking). By this alternative of cooperation control (flat transaction management), the design transaction inherits neither locks nor recovery information from committing remote operations. In this case, long duration locks (e.g. CHECKOUT locks) are not controlled by the database system but must be managed by the users themselves (for instance, in the form of tuples of some special relation as the OBJECTLOCK relation proposed in [LoPl83]).

The cooperation between server and workstation can also be realized on the basis of a distributed nested transaction mechanism as , for instance, the one proposed in [HäRo87] which follows the nested transaction paradigm of [Moss81]. When the design transaction starts its first remote operation at the server node, the public system becomes aware of its existence and creates a so-called agent transaction for the design transaction at that node. The public system considers all remote operations started by the design transaction as sub-transactions of its agent at the server node. That is, the agent inherits results, locks, and recovery information from all remote operations which commit. If the design transaction aborts, the public system aborts its agent at the server node. Recovery information is used to restore the original state of the public database, locks held by the agent are released, and the agent is deleted. If, on the other hand, the design transaction commits at the workstation, the public system commits the agent at the server node and releases its locks in the public database. Thus, results of committed CHECKIN operations and remote queries are made public at the server node, only if the corresponding design transactions commit.

2.7.5 The Distributed Version of the Reference Architecture

The decisions about the distribution of our reference DBS architecture over a server-workstation computer system rely on the distribution aspects discussed above and are presented below.

Regarding the distribution of the database software over the various processing nodes, we chose the alternative followed by the PRIMA project [HHMM88]. Both kernel layers (i.e.

L1 and L2) would be realized at the server node, while the application-oriented layer (i.e. L3) would be implemented at the workstation. As with PRIMA, we would not realize a copy of the kernel system at the workstation to control the local database. A special system layer (i.e. the object-supporting layer) would be constructed between L3 and L2, instead. This extra layer should support L2´s data model at the workstation (for instance, by supporting a local version of the object buffer). Moreover, the object-supporting layer would also support the communication with the kernel system at the server node. Figure 1.4 depicts the distribution of the reference DBS architecture over a server-workstation network.

Although multi-level strategies for the cooperation between server and workstation seem to be more appropriate in terms of data communication load, we feel that the realization of multi-level cooperation strategies could possibly degrade system performance because of the mapping activities associated to them. However, we have implemented a simulation model on the basis of our reference architecture that permits the analysis of single-level and multi-level cooperation strategies. Within the context of this work, we have simulated only a single-level cooperation strategy at the complex object level, though.

Since the integration of information systems (as shared databases, computer networks, etc) is a reality already, we feel that design database systems will have to support the coexistence of long-duration and (conventional) short transactions to cope with different database applications which will process related data in the same enterprise.

In the following chapters, we discuss further distribution aspects concerning dynamic properties of the reference model derived here (e.g. transaction management).

Chapter 3

Transaction Models for Design Applications

In this part of the dissertation, we review five of the best known design processing models which have been proposed in the literature and use them as a basis for the classification and generalization of design transaction models. Before describing the models, though, we discuss the main reasons why design transactions cannot be realized as conventional transactions, that is, why the (conventional) transaction paradigm cannot cope well with all the processing requirements of design applications.

A transaction is defined as a unit of work to be carried out by the DBS [Date83] or as an execution of a program that accesses a shared database [BeHG87]. In fact, the transaction paradigm as described in [Gray78] represents a special unit of work, namely one which guarantees that user actions either bring the database to a consistent state [Gray80] or are not executed at all. Besides of being atomic and guaranteeing database consistency, the conventional transaction represents the unit of work isolation in multiprogramming environments, and its results must survive failures if the transaction commits. Therefore, the transaction represents, at the same time, the units of atomicity, isolation, consistency, and durability of transaction-oriented processing environments.

The transaction model as described above copes well with the characteristics of user work in so-called business-related database applications where transactions are typically of short duration (i.e. terminate in a few seconds). For short transactions, it is acceptable that they are completely backed out in case of failures, or that they are blocked during the execution of other transactions which access common data.

The conventional transaction model is not suitable to represent the design transaction in design applications such as CAD/CAM or software engineering, though. As already stated at the beginning, user work in design applications can span longer periods of time. Complete rollback would then be unacceptable. Moreover, these applications are characterized by cooperative work among users. The isolation property of transactions should, therefore, be relaxed in some cases.

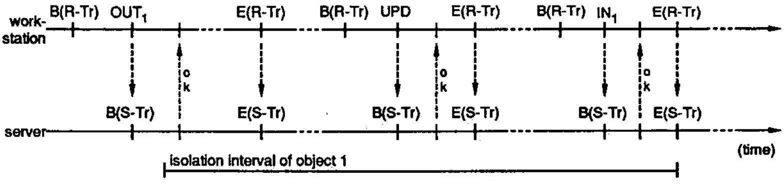
To support the above mentioned and yet other requirements of design applications, new processing models have been proposed which respectively associate the properties of the conventional transaction with different units of user work. Since the new processing models are partially based on the transaction paradigm and aim at representing larger units of work in design applications, they are conventionally called design or long transaction models.

By the description of the processing models which follows, we will distinguish several transaction types. Conventional transactions which are executed at the server node will also be called short transactions (S-Tr), because they represent relative short units of work (e.g. single database operations) which are executed by the public system on behalf of the designer at the workstation. On the other hand, conventional transactions which are executed at the workstation will also be called recovery transactions (R-Tr) because, in most cases, they solely represent a unit of recovery for the processing models. Other (sub)transaction types will be introduced during the description of the design transaction models which follows. Since most of the processing models to be reviewed here do not have a specific name, we will identify them through their main characteristics.

3.1 Isolated CHECKOUT/IN Operations

The isolated CHECKOUT/IN transaction model (TM1) has been first proposed in [HaLo81]. This model is perhaps the simplest one and basically relies on the conventional transaction paradigm. Figure 3.1 illustrates the dynamical characteristics of the isolated CHECKOUT/IN transaction model.

The designer processes data objects in his local, private database at the workstation. The local database system controls the user work at that processing node. When the user wants to process an object which is not present in the private database, he starts a remote (sub)transaction at the server node that executes a CHECKOUT operation. Similarly, when the user wants to commit his changes on an object (or insert/delete an object), he checks it back into the public database (through another remote transaction).



S-Tr: short trans.; R-Tr: recovery trans.; B(Tr): begin trans.; E(Tr): end trans.; OUT: checkout; IN: checkin; UPD: short update operation.

Fig. 3.1: Isolated CHECKOUT/IN transaction model (TM1)

The fact that the designer may sometimes need to manipulate data directly in the public database (e.g. queries and short update operations on sets of tuples or records) is not directly modeled by TM1. One can easily conclude, though, that the designer must start (sub)transactions in the public system to execute those operations, too.

In the isolated CHECKOUT/IN model, the notion of design transaction is known neither to the public system nor to the local system. Both systems execute conventional transactions. The execution of transactions which spawn sub-transactions at the public system can be controlled either by means of some version of the two-phase commit protocol [MoAb86] or through the realization of the nested transaction concept [Moss81]. In Figure 3.1 as well as in all other figures of this chapter which depict dynamic properties of design processing models, we suppose that transactions which span processing nodes follow a two-phase commit protocol. That is, for simplicity reasons we assume a flat distributed transaction mechanism in the models to be explained. This can be clearly depicted from the figures. See, for instance, Figure 3.1. By receiving a "start CHECKOUT" message from the workstation (which is part of a recovery transaction there), the public system starts a short transaction (*B(S-Tr)*) at the server node to copy the desired data from the public database into the private database at the workstation. By finishing this task, the public system sends the workstation an "ok" message and waits for a reply to either abort or commit the short transaction *(E(S-Tr))*.

Since conventional transactions release all their locks at transaction end, CHECKOUT locks cannot be automatically managed by the public system. They are long-duration locks that must be held even after the CHECKOUT sub-transaction commits. The designers must maintain a global CHECKOUT lock file in the public database, instead. This file must be updated by the CHECKOUT/IN sub-transactions themselves. Thus, the designers are responsible for the control of both long-duration locks and the system's CHECKOUT lock file. On the one hand, this solution can lead to a better synchronization of the design work, since the designers have a better knowledge about who may access objects concurrently and when it can be done. On the other hand, the public system cannot automatically guarantee any synchronization protocol (e.g. serializability) anymore.

Furthermore, the isolated CHECKOUT/IN model does not relate the actions of the same design transaction (i.e. the actions which are started by the same designer) on different public objects. Conventional transactions represent the only units of atomicity and durability at both the public and the local systems. Thus, it is, for instance, impossible for the database system to automatically rollback the whole designer's work.

If no objects have been checked back into the public database yet and no short update transaction has been executed on his behalf at the server node, the designer can back out his whole work by simply deleting his local, private database and releasing locks in the CHECKOUT lock file of the public database. Otherwise, he must start compensation (sub)transactions in the public system to rollback possible CHECKIN and short update operations at the server. The task of rolling back committed CHECKIN operations can be simplified if the public system treats updated objects as new object versions as it is, for instance, proposed in [KSUW85]. In this case, the designer can back out CHECKIN operations by deleting the new object versions created in the public database.

Problems may arise, though, if other users have already copied the new created versions into their private databases. TM1 does not prevent the occurrence of the so-called cascading-abort effect [BeHG87]. This effect can take place when transactions can see temporary or incomplete results of other transactions. These results can be considered to be consistent only if the transaction which created them commits. In a conventional data processing environment (e.g. business environment) which relies on transaction serializability, the recovery mechanism would back out all transactions which accessed temporary results of a transaction being aborted. In the design environment, this could represent a huge waste of time and work done. A less drastic solution would be to inform the transactions that the object versions they manipulate have become invalid. The designers, then, have a chance to decide whether they can save some of the work done or must roll back their entire transactions. Furthermore, if transaction serializability is to be enforced, the system or the designers themselves should avoid that design transactions which see temporary results commit before the transactions which generated those results.

For TM1, the conventional transaction also represents the unit of isolation for queries and short update operations in the public system. By object processing activities, the unit of isolation in the public database is represented by their respective CHECKOUT/IN time intervals.

In the literature, no special synchronization mechanism at the workstation is presented for TM1. For local systems which can process recovery transactions concurrently, it is assumed that conventional synchronization techniques (e.g. short-duration locking) will be implemented. In this case, the unit of isolation at the workstation would also be represented by the conventional transaction.

As with most of the other design transaction models, explanations about how and when to check as well as to enforce design consistency by TM1 cannot be found in the literature.

Since every short transaction at the public system as well as recovery transaction at the local system represents only a part of the overall design work, it might be impossible to enforce design consistency either in the public database or private database at the end of every one of these transactions. In most systems, design consistency will be tested and enforced by the application layer through the execution of test procedures (which can, for instance, be realized through event/trigger mechanisms [Kotz88]). Since the decision of how and when to test design consistency is taken in higher system levels, we do not associate the concept of short or recovery transactions with these tasks. Thus, we conclude that short and recovery transactions can only guarantee database consistency for lower levels of abstraction (e.g. record-oriented level [HäRe83]). By the isolated CHECKOUT/IN model, design consistency is considered to be responsibility of the application and should be enforced, at the latest, at the end of the overall design work.

On the basis of the description above, we can conclude that TM1 does not fully support the notion of a design transaction. The public system sees the design work as a set of independent processing steps on different objects. Moreover, no features are presented to support the hierarchically structured design environment. That is, designers cannot exchange non-committed results or commit results only for some group of other designers. The next design processing model to be presented can be seen as an extension of TM1 that captures the notion of design transaction in the public database.

3.2 Related CHECKOUT/IN Operations

The related CHECKOUT/IN model (TM2) has been described in [DAM88a] and implemented in the DAMOKLES database system. TM2 assumes the same design environment as TM1. What distinguishes TM2 from TM1 is that the former introduces the notion of a design transaction (D-Tr) at the public system. Figure 3.2 depicts the dynamical characteristics of TM2.

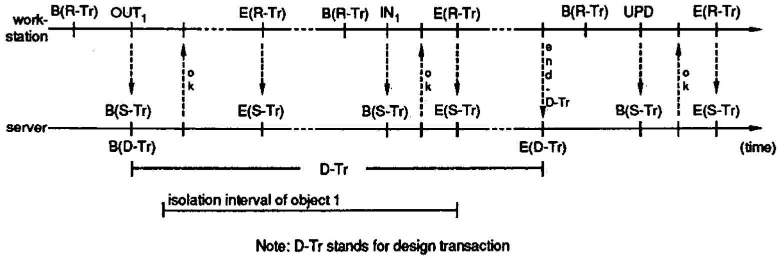


Fig. 3.2: Related CHECKOUT/IN transaction model (TM2)

The public system starts a D-Tr on behalf of the designer either by request or when it receives the first remote operation from the workstation. Objects are processed by the designer at the workstation. Every time the designer needs to process an object which is not present in his local, private database, he starts a (sub)transaction at the public system which checks the wanted object out. All those CHECKOUT/IN operations executed at the public system on behalf of the designer are considered part of the his D-Tr. On the other hand, queries and short update operations in the public database that might be started by the designer are not modeled by TM2 at all.

D-Tr can be viewed as the top-level transaction of a two-level nested transaction hierarchy. D-Tr holds CHECKOUT locks for objects which are manipulated by recovery transactions at the workstation. These locks are released when the associated objects are checked back into the public database. The designer terminates his work at the public system by executing a COMMIT D-Tr operation. When this occurs, the public system verifies if all objects checked out have already been checked back in. If this test yields true, the system commits D-Tr; otherwise, the designer receives a corresponding message and D-Tr is kept active.

Although the public system can use D-Tr to automatically guarantee object isolation even after the (sub)transactions which have executed CHECKOUT operations commit, it cannot use D-Tr as a unit of atomicity for the design transaction, since the designer may check objects back into the public database at any time. Thus, all those problems related to recovery in TM1 are also present in TM2, namely the recovery mechanism must cope with the cascading-abort effect and cannot provide for automatic design transaction backout.

At both the public and the local systems, D-Tr represents the unit of consistency for the overall design work. Although this aspect is not discussed in the literature, it is easy to understand that database consistency at lower abstraction levels can already be enforced in the public database by CHECKIN. On the other hand, overall design consistency can only be tested and guaranteed at the end of D-Tr.

As by TM1, the conventional transaction represents the unit of atomicity and durability in the public system as well as in the private system. It could also be used as the unit of isolation for queries and short update operations at the server node. The unit of isolation for object processing is represented by the CHECKOUT/IN time interval.

Since TM2 does not model data processing at the private system in detail, it is not certain if a work unit for isolation is needed for that system. It would only be true, if recovery transactions could run in parallel at the workstation. As with TM1, cooperative work among designers is not modeled by TM2, either. Therefore, design results must first be checked into the public database (and made accessible to all designers), before another designer can copy them into his private database.

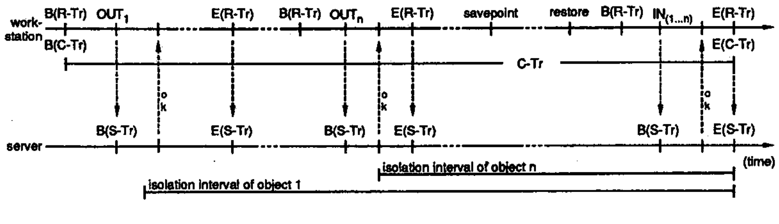
3.3 Conversational Design Transactions

The conversational design transaction model (TM3) has been presented in [LoPl83]. It assumes the same system configuration and DBMS architecture as TM1 and TM2. TM3 can also be understood as being an extension of TM1.

TM3 introduces the notion of a conversational transaction (C-Tr) in the local, private database. C-Tr represents the overall design effort of the designer and is comparable to D-Tr in TM2. There are some important differences between both design transactions, though. While D-Tr exists in the public system, C-Tr is controlled by the local system at the workstation. Furthermore, TM3 models the design transaction in such a way that updated objects are accessible to other designers only if C-Tr commits. Conversational design transactions follow a strict two-phase CHECKOUT lock protocol.

Figure 3.3 depicts the dynamical characteristics of C-Tr. The local system treats the user work as a (long) design transaction (i.e. a C-Tr). From inside his C-Tr, the designer may start a set of recovery transactions which process objects in the local database. When the designer needs to get an object from the public database, he starts a short (sub)transaction in the public system which executes a CHECKOUT operation.

Since queries and short update operations in the public database are not modeled by TM3 (as it is the case by TM1 and TM2, too), these operations are not considered part of C-Tr. Moreover, the only way to integrate them into the conversational transaction model is to represent them as remote (sub)transactions at the public system.



Note: C-Tr stands for conversational transaction

Fig. 3.3: Conversational transaction model (TM3)

All design objects which are checked out of the public database for update as well as those created during the design work are checked back into the public database through a global CHECKIN operation which is executed at the end of C-Tr. Thus, at most one CHECKIN operation can be executed from inside a conversational transaction.

In [HHMM88], the conversational design transaction model has been expanded and ideas of how to implement it have been presented. Since some of the new properties which have been added to the model can influence recovery activity, they will be presented here, too. According to the expanded version, designers can issue other commands from inside the C-Tr besides starting recovery transactions: SAVEPOINT, RESTORE, SUSPEND, and RESUME. The SAVEPOINT command forces the actual state of C-Tr to be saved. The RESTORE command rolls back C-Tr to a previously generated savepoint. The SUSPEND command creates a savepoint for C-Tr and (temporarily) terminates it. The RESUME command restarts C-Tr at the last SUSPEND savepoint issued.

For TM3, the conventional transaction represents the unit of atomicity and durability in the public system. CHECKOUT/IN time intervals, in turn, represent the unit of isolation for object processing. C-Tr also represents the unit of design consistency in the public database. Since all object updates processed by the designer during his C-Tr are checked into the public database at once, all necessary design consistency tests can be executed inside the (sub)transaction which executes the CHECKIN operation at the public system.

The recovery transaction represents the unit of atomicity in the private system. It also provides for durability in case of system failures at the workstation. The effects of committed recovery transactions do not survive, though, if the designer rolls back his C-Tr. Results of committed recovery transactions can also be made invalid, if the state of the C-Tr is restored to a previously generated savepoint. Besides, C-Tr represents the unit of design consistency for the private system. The designer can back out C-Tr completely by deleting his private database and releasing associated CHECKOUT locks in the public database.

The conversational transaction model makes design transactions appear atomic at the public system. Thus, the recovery mechanism need not support cascading aborts. Since TM3 does not take short update operations in the public database into consideration, the designer can only invalidate their results by executing the respective compensation transactions at the public system.

If conversational transactions are realized as nested transactions (as it is suggested in [HHMM88]), it is possible for the public system to automatically control long-duration locks in the public database, since conversational transactions, then, inherit locks and recovery information from committing sub-transactions which execute CHECKOUT operations on their behalf. On the other hand, if a distributed flat transaction management is realized, the public system does not become aware of the existence of the C-Tr at the workstation and the designers must manage CHECKOUT locks by themselves, as it is the case with TM1.

Although TM3 represents an evolution towards modeling design transactions when compared to TM1 and even to TM2, this model does not capture some important characteristics of the design work, namely the cooperation among designers and details of the way objects are processed at the workstation. These characteristics are taken into consideration by the next two design transaction models to be presented.

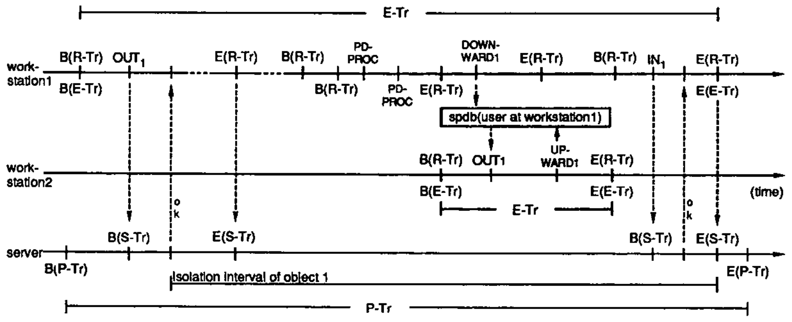
3.4 Engineering Transactions

The engineering transaction model (TM4) has been proposed in [KLMP84] and refined in both [BaKK85] and [KoKB87]. It relies upon three major concepts: nested transactions, semi-public databases associated with nested transactions, and the notion of a design transaction consisting of a set of possibly concurrent, conventional transactions at the workstation.

TM4 introduces the notions of project transaction (P-Tr), engineering or client/subcontractor transaction (E-Tr), and semi-public database. P-Tr characterizes the efforts of a group of designers while E-Tr represents the design efforts of a single designer. Project transactions run at the public system and logically partition the public database. Each P-Tr is the root of a hierarchy of nested E-Tr. P-Tr acquires and releases locks in the public system for all its descendants. Project transactions follow a long two-phase locking protocol.

Figure 3.4 depicts dynamic properties of TM4. Engineering transactions are started and executed by designers at workstations. Each E-Tr is associated with a private database (managed by the local, private system) and a semi-public database. It is subdivided into a set of recovery transactions. Concurrent R-Trs follow a short two-phase locking protocol. By modeling concurrent recovery transactions at the workstation, TM4 tries to better capture the real processing mode of design transactions in the private system. Usually, the designer starts design-tool programs at the workstation that may execute in parallel and even start other programs or external subroutines. The execution of each tool-program is modeled by TM4 as a recovery transaction. Thus, recovery transactions may run in parallel, start (sub)transactions, and access data concurrently.

Through E-Tr, the designer processes objects in his private database. To make semi-committed object updates accessible to a selected group of designers (e.g. designers of his own group), he moves the respective updated objects from his private database into his semi-public database and grants the selected designers specific access rights. This operation is called DOWNWARD COMMIT. The authorized designers, then, may copy the semi-committed results into their own private databases through the execution of CHECKOUT operations at the system which controls the semi-public database where the wanted objects are located. CHECKOUT/IN operations started from the same E-Tr follow a long two-phase protocol. That is, the E-Tr (i.e. the designer) may not execute CHECKOUT operations anymore after it(he) has already executed some CHECKIN operation.



P-Tr: project trans.; E-Tr: engineering trans.; DOWNWARD: downward commit object; UPWARD: upward commit object; PD-PROC: process object in the private database; spdb: semi-public database.

Fig. 3.4: Engineering transaction model (TM4)

Each E-Tr may only have one parent (that is, one immediate ancestor transaction). E-Tr's parent may be either another E-Tr or the P-Tr itself. Each E-Tr may check out objects of the semi-public database of, at most, one other E-Tr. The former becomes a child of the latter. If E-Tr only checks out objects of the public database, it becomes a direct descendant of P-Tr (in this case the public database can be viewed as P-Tr's semi-public database). Through the semi-public database of its parent, though, E-Tr may indirectly check objects out of the semi-public database of any ancestor.

To commit his E-Tr, the designer must check all processed objects back into the semi-public database of E-Tr's parent. The parent inherits all locks held by E-Tr (i.e. CHECKOUT locks as well as other locks). If E-Tr aborts, it releases all locks it acquired. In this case, E-Tr's parent inherits only those locks which it already had owned, before E-Tr has inherited them. The other locks are directly inherited by those ancestors from which the locks were inherited. In the literature, no explanation is given about what happens with descendant E-Trs when any of their ancestors is backed out. It would be reasonable to think that they would, at least, be asked to release those objects which relate them to their ancestors.

If we suppose that all semi-public databases are located at the server node and managed by the public system as it is proposed in [KLMP84], we can derive following observations about transaction properties in TM4. P-Tr represents the whole work of a group of designers and can be seen as the unit of consistency for this work. P-Tr also isolates this work from those of other groups of designers. By backing out P-Tr, the public system must rollback all active as well as committed E-Tr which are descendants of P-Tr.

E-Tr models the design transaction. It represents the unit of consistency of the designer's work at the public system as well as at the private system. E-Tr also represents the unit of isolation for all data processed by the designer during his work. If E-Tr does not follow a strict two-phase protocol, the public system should protect the results of each CHECKIN operation in case of system failure at the server node. Otherwise, E-Tr is also the unit of durability for the public system.

The only work unit which must be executed atomically at the public system is the conventional transaction. This transaction can execute either CHECKOUT/IN operations or short update operations in any of the databases controlled by the public system. At the private system, the recovery transaction represents the unit of atomicity, isolation, and durability against system failures. Committed R-Trs can be backed out, though, if the designer decides to rollback his E-Tr.

TM4 is the only model to be reviewed which takes into consideration that the designer may want to process other data besides design objects from inside his E-Tr. For instance, the designer may want to update some relation in the public database. TM4 treats all public data in a homogeneous form. Thus, the recovery mechanism can support automatic backout of E-Tr and P-Tr even if designers manipulate other data besides objects.

The action of backing out an engineering transaction can generate the cascading-abort effect in two directions. E-Tr's descendants as well as ancestors must, at least, be informed about the decision to abort it. Descendants would have to give objects back to E-Tr (or release the corresponding locks). In the worst case, they would also abort. Likewise, ancestors which have already got objects back from the aborting E-Tr would have to decide if they can continue or must abort.

3.5 Group Transactions

The group transaction model (TM5) has been proposed in [KSUW85] and relies on the concepts of multi-level transaction management and object version graphs. TM5 assumes that objects are versioned and that the public system manages a global version graph for each object in the public database.

TM5 introduces the concepts of group transaction (G-Tr) and user transaction (U-Tr) which can respectively be compared to the concepts of project and engineering transactions. While G-Tr represents the design efforts of a group of designers, U-Tr characterizes the efforts of one specific designer (i.e. it represents the design transaction). G-Tr is the top-level of a two level transaction hierarchy in which user transactions appear as sub-transactions. Locks acquired by G-Tr can be inherited by its sub-transactions. As opposed to TM4 that associates a semi-public database with every design transaction, TM5 associates a group database with each group transaction. The user transaction is related only to its private databases (the so-called user database).

Figure 3.5 presents the dynamic characteristics of TM5. Objects are checked out of the public database by G-Tr and placed in the group database, where they are integrated into local version graphs. U-Tr checks objects out of the group database and places them in its private database. Updated objects are first checked back into the group database and, when no other U-Tr needs them anymore, into the public database.

G-Tr acquires locks on objects in the public database while U-Tr locks data objects in the group database. While G-Tr follows a two-phase CHECKOUT/IN lock protocol in the public database, U-Tr is not requested to follow a two-phase protocol in the group database. That is, U-Tr may execute CHECKOUT/IN operations in any order. Unfortunately, no detailed explanation about the internal structure of user transactions can be found in the literature. TM5 does not model the design work at the private system.

TM5 models cooperative work among design transactions by introducing four commands which can be executed from inside U-Tr. There exist two alternatives to exchange semi-committed results. A designer can lend an object version which he checked out of the group database to another designer by executing the GRANT command. The other designer copies the object version into his private database by executing a FETCH command. The designer who borrowed the object version gets all rights over it, except the right to check it back into the group database. He must give the object version back to its owner. This must occur before he terminates his U-Tr. Designers must execute a RETURN command to give the rights over a borrowed object version back to its owner. The owner copies the version back into his private database by executing another FETCH command. The designer is not allowed to terminate his U-Tr, before he gets back all object versions he has granted.

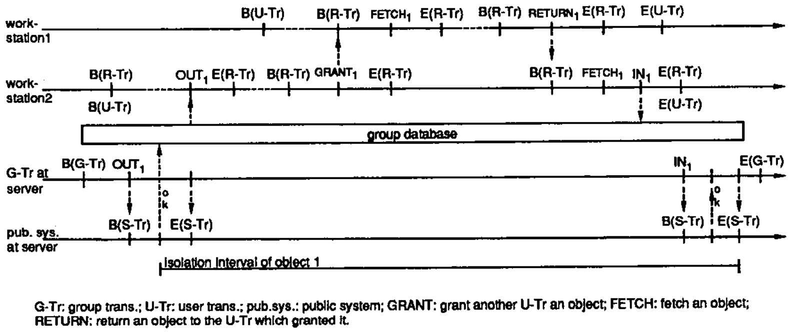


Fig. 3.5: Group transaction model (TM5)

To exchange object versions which are not expected to be returned, designers can use the PASS statement. As with the GRANT command, the object being passed is copied into the database of the receiver by means of a FETCH command. In contrast to the GRANT command, the designer who receives the object can consider it as being his property and need not return it to it´s original owner.

At the public system, G-Tr represents the unit of consistency for the whole work of the group of designers it represents. If G-Tr follows a strict two-phase lock protocol, it is also the unit of isolation in the public database. Otherwise, this property is associated with the CHECKOUT/IN time intervals during which object versions are copied from the public database, processed by design transactions of some group transaction, and reintegrated into the public database once again. CHECKOUT/IN operations must be executed atomically and their results must survive failures at the public system.

Since U-Tr need not follow a two-phase lock protocol, consistency in the group database cannot be based on transaction serializability. Therefore, another correctness criterion for concurrent U-Trs must be applied to the group transaction environment.

At the private system, the unit of design consistency is the U-Tr. The unit of isolation is represented by the design work which is done from the moment the object version is checked out of the group database until it is either checked back into it or granted/passed to another designer. Since U-Tr is a long-duration transaction, it would be very dangerous to use it as the unit of atomicity and durability at the private system. Thus, we feel that by any reasonable implementation of TM5, U-Tr would consist of a set of recovery transactions.

3.6 Generalizing the Design Processing Models

3.6.1 Main Characteristics of the Design Processing Models

All design processing models reviewed above try to capture the characteristics of the design environment and represent them in the database system. Basically, these models differ from one another to the extent in which they model the design environment.

While TM1 models only the object processing activity, TM2 and TM3 support the design transaction concept which represents the complete activity of the designer and consists of a set of related object processing steps. Finally, TM4 and TM5 also model the cooperative work of a group of designers through the project and group transaction concept, respectively. P-Tr as well as G-Tr consist of a set of related design transactions which may exchange non-committed object updates.

Figure 3.6 relates the processing models to the hierarchy of work units in the design environment. These work units represent spheres of control in the database system. Therefore, they can be modeled as transactions. Since these transactions form a hierarchy, they can be represented as nested transactions, too.

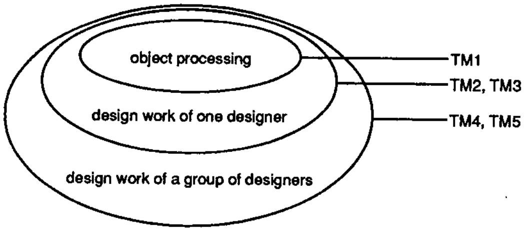

object processing
design work of one designer
design work of a group of designers
TM1
TM2, TM3
TM4, TM5

Fig. 3.6: Relating the transaction models to the design work hierarchy

In the following, we present a list of the main characteristics of the design environment which are captured by the various design processing models.

- The database is basically subdivided into a public database and a number of private databases. The public database stores the released versions of design objects and, maybe, other data related to the overall design enterprise. The designers work on design object copies (i.e. non-released object versions) which are kept in private databases. The designer may only integrate a design object copy into the public database (either as a next released version of the object or as its only valid version) when this copy is design object consistent. These characteristics of the design environment are modeled by all processing models reviewed.
- The designer copies data from/into the public database on an object basis. That is, complex/molecular objects are copied as a whole from/into the public database. Consequently, the design object represents a possible granularity unit for concurrency control (and recovery) purposes at the public system. This characteristic of the design environment is modeled through the CHECKOUT/IN operations and supported by all processing models reviewed.
- The set of related work steps executed by the designer constitutes the design transaction. This transaction represents a sphere of control in the design environment. All models but TM1 model the design transaction concept. They do that in different ways, though. While TM2 uses the design transaction concept only to support

automatic CHECKOUT-lock management in the public database, TM3, TM4 and TM5 model the design transaction as a unit of work at the private system which can even be backed out by the designer.

- Besides relating object processing steps to each other at the private system, the design transaction may also relate them to queries and short update operations which are started by the designer at the public system and directly manipulate data in the public database. Only TM4 models queries and short update operations in the public database as part of the design transaction.
- Since the design transaction consists of a set of processing steps and can take long, it should be possible to protect parts of it from system failures. TM3 and TM4 model this characteristic of the design environment explicitly. Both models represent processing steps as short-duration transactions which automatically guarantee atomicity and durability in case of failures at the private system. Besides that, TM3 also models the concept of savepoint which enables the designer to save specific design states which he may want to restore some time later (SAVEPOINT/RESTORE statements).
- Reflecting dynamic characteristics of design tools, the private system should support the concurrent execution of related processing steps. Moreover, it should allow processing steps to be started by other processing steps (to model, for instance, design activities in window-supporting systems). The design transaction, then, could consist of a set of parallel, possibly concurrent processing steps. From the models reviewed, only TM4 represents the design transaction in this way.
- Design cooperation is another characteristic of the design environment. Usually, a group of designers cooperate in the same design work. In this case, semi-consistent design parts may have to be exchanged among designers of the same group. TM4 and TM5 model the concept of group transaction which relates design transactions to a higher sphere of control in the DBS. The group transaction can be viewed as the root of a nested transaction hierarchy of design transactions. TM5 permits only two-level transaction hierarchies. TM4 permits n-level hierarchies where each design transaction can represent the design efforts of more than one designer (i.e. the efforts of the owner of the design transaction and those of designers which execute descendant design transactions).

3.6.2 Generalizing the Processing Models Reviewed

Based on the design processing models reviewed and the hierarchical structure of the design work, we can derive three simplified design processing models which generalize those found in the literature. Each one of the new models represents a specific class of design processing models and will be described on the basis of the properties shown by the models reviewed.

In the first general model (GM1) the public system views the work of the designer as a set of independent object processing steps. These steps are represented by processing activities executed during CHECKOUT/IN intervals. The notion of design transaction is used at the public system only to enable the realization of long-duration locks. In the second model (GM2), the public system views the work of the designer as a (design) transaction which executes at a remote processing node. The design transaction follows a strict two-phase CHECKOUT/IN protocol to copy objects from and to the public database. The third processing model (GM3) models the design cooperation of a group of designers.

All three models represent the design work at the private system in the same way. The whole activity of the designer is captured in the notion of a design transaction at the workstation. This transaction is subdivided into a set of possibly concurrent, recovery transactions. Moreover, these recovery transactions support what we call an internal savepoint mechanism. Since the recovery transaction can sometimes take longer than conventional transactions in business-related applications (e.g. when the designer creates his design using a graphic tool), the designer may want to save some internal state of it either to protect it from system failures or to restore this state later on, if he thinks it is necessary. We make two restrictions to internal savepoints, though. First, they cannot be generated as long as there exist active sub-transactions and/or remote operations which were started by the recovery transaction. Secondly, they can only be restored by the user from inside of the same recovery transaction where they were generated. These two restrictions simplify savepoint restoration activities in recovery transaction hierarchies (see [HäRo87]). They help defining a clear scope of recovery for savepoint restoration.

At the public system, remote operations are modeled as short transactions by all general processing models. Thus, CHECKOUT/IN operations (in the public or group database) as well as queries and short update operations started by the designer at the workstation are executed as isolated short transactions at the public system. In GM2 and GM3, these transactions are logically related to each other by means of the design transaction.

3.6.2.1 The Design Transaction in GM1

At the beginning of his work, the designer starts a design transaction at the private system. When this transaction executes its first CHECKOUT operation, the public system notices its existence and starts controlling its execution in the public database. The design transaction inherits CHECKOUT locks from conventional transactions which execute CHECKOUT operations on its behalf at the public system. These locks are released when the objects are checked back into the parent database. Short-duration locks acquired by the execution of queries and short update operations are released at the end of the respective short transactions. Thus, these operations are not associated with the design transaction.

In GM1, the design transaction may execute CHECKOUT/IN operations in any order. Moreover, objects which have been checked out must not be checked in atomically (i.e. in only one CHECKIN operation). Therefore, the DBS cannot guarantee that the concurrent execution of design transactions is always serializable. Consequently, design transaction schedules may become non-recoverable [BeHG87]. Furthermore, the designer cannot abort his design transaction. He can at most start compensation operations for committed recovery transactions.

When the designer wants to commit his design transaction, the private system must consult the public system. Design transactions are allowed to commit only if all objects they have checked out of the public database have already been checked back in there.

3.6.2.2 The Design Transaction in GM2

In contrast to GM1, GM2 enforces a strict two-phase CHECKOUT/IN protocol for design transactions. That is, all objects checked out by the design transaction are checked back into the parent database at transaction commit. Moreover, this operation must be atomic. Thus, every design transaction executes, at most, one CHECKIN operation at the public system. The design transaction constitutes the unit of isolation at the public system concerning object processing activities. As opposed to GM1, GM2 enforces recoverable design transaction schedules at the public system. Recovery can, therefore, be based on serializability to maintain database consistency.

Besides modeling internal savepoints for recovery transactions, GM2 introduces the concept of external savepoints for design transactions. This savepoints save particular states of the design transaction. These states can be restored by the designer at any time before the design transaction terminates. To simplify savepoint restoration, we restrict external savepoint generation to those points in time when no recovery transaction is active at the private system.

3.6.2.3 Modeling Cooperative Group Work in GM3

GM3 models the cooperative work of a group of designers as a group transaction at the public system. Similar to the group transaction of TM5, each G-Tr is associated with a group database, respectively. As with G-Tr, the group database is also located at the server node and controlled by the public system. We could imagine the group database being implemented as a subset of the public database that can grow dynamically. G-Tr is the parent transaction of all design transactions belonging to designers of a group.

Design transactions access data in the public database through the group transaction. That is, design transactions check objects out of the group database. If the desired objects are not there, G-Tr checks them out of the public database. Similarly, design transactions check objects back into the group database after having processed them. Furthermore, locks acquired by queries and short update transactions in the public system are inherited by G-Tr.

While G-Tr follows a (long) strict two-phase locking protocol and a strict two-phase CHECKOUT/IN protocol at the public system, design transactions follow a predicatewise two-phase locking protocol, as it is presented in [KoKB87]. To enable designers to exchange results before design transaction commit, database consistency cannot be based on transaction serializability. In [KoKB87], the requirement of transaction serializability is relaxed by replacing it with a requirement on preservation of the consistency constraint. We briefly explain this concept below.

Database consistency can be described by a consistency constraint. The consistency constraint for the database can be described in the form of a predicate. A transaction is said to preserve database consistency if it preserves the consistency constraint when it runs alone. The consistency constraint for the database can be expressed as a conjunction of relatively simple consistency constraints for parts of the database. Since most transactions only process parts of the database, they must only preserve some of the predicates that form the consistency constraint for the whole database.

It is possible to represent every consistency constraint in the conjunctive normal form, that is, as a conjunction of simpler predicates such that none of them contains any **ands**. Each such a predicate will be called a conjunct.

We can, then, write the consistency constraint for the whole database as a set of conjuncts related by **and** operators. Moreover, we can subdivide the database itself into a set of groups of data where each group is, respectively, related to only one of the conjuncts of the consistency constraint. Thus, the invariant of each transaction can be represented by the conjunction of the conjuncts related to the data it manipulates. If we divide the database in such a way that each group of data corresponds exactly to one design object, each design object will be related to one of the conjuncts of the consistency constraint for the database.

The relation between consistency constraints and groups of data is used in [KoKB87] to synchronize concurrent execution of design transactions on an object basis. The predicatewise two-phase locking protocol observes two-phase locking only with respect to each group of data being manipulated. That is, data of a group cannot be locked by a transaction if it released some lock on other data of the same group, already. On the other hand, data objects of different groups can be locked and unlocked independently.

In the following, we assume that the design database can be subdivided into a set of design objects, each of which being related to exactly one conjunct of the consistency constraint for the whole database. Moreover, we further suppose that every conjunct of the constraint is related to some design object of the database.

In GM3, design transactions exchange object updates by simply checking objects back into the group database. They can do that at any time. They are only required not to check out the same object more than once during normal execution. As with GM2, the designer can generate external savepoints for his design transaction. He can, then, roll back design work by either backing up D-Tr to some previously generated external savepoint or by backing it out completely.

3.6.3 Properties of the General Design Processing Models

Since the processing models derived above can be considered as being extensions of the conventional transaction paradigm, we can identify transaction properties for them, too.

While the transaction paradigm relates all its properties (i.e. atomicity, isolation, correctness, and persistency) to the same unit of user work, namely the conventional transaction, the design processing models relate these properties to different units of work (e.g. recovery transaction, design transaction, group transaction). In this section, we relate units of work that are modeled by the general processing models presented above to properties that must be guaranteed by the recovery mechanism. Since we are mainly interested in investigating database recovery requirements in the design environment, we will concentrate our efforts on the study of those properties of the design work that can influence recovery activity. Therefore, we will not further investigate units of consistency in the design environment.

For each one of the general models presented above, we will discuss atomicity, persistency in case of failures, and two other properties. The first of them is related to either units of work or control information which should survive failures even before the design work terminates. Contrary to the conventional transaction paradigm, the new processing models enable designers to save internal transaction states that must be protected against failures and should be restored on user request. Furthermore, the state of longer transactions (i.e. design transactions and group transactions) must also survive failures. In the following, this property will be called **temporary persistency** (to relate it to and differentiate it from the persistency property of conventional transactions).

The second property to be introduced is related to the set of work units that can be backed out by the designer (before their results are committed). In the conventional transaction model, this set contains only one element, namely the conventional transaction itself. The new models allow designers to roll back other units of design work as well (e.g. design transaction and group transaction). Figure 3.7 summarizes what is described below.

3.6.3.1 Transaction Properties in GM1

In GM1, (conventional) short transactions represent the unit of atomicity and durability at the public database. GM1 models the design transaction at that system only to relate it to its associated CHECKOUT locks. Thus, D-Tr represents neither a unit of atomicity nor a unit of durability at the public system. On the other hand, the design transaction represents the unit of temporary persistency at that system. That is, the state of the design transaction (e.g. information about its associated CHECKOUT locks) must survive system failures at the public system.

At the private system, only (conventional) recovery transactions for which no internal savepoint has been generated must be executed atomically. On the other hand, recovery transactions for which, at least, one internal savepoint has been generated must survive system failures at the workstation even if they have not yet committed. In case of failures, the recovery manager must restore their youngest internal savepoint. Thus, the design work executed between recovery transaction begin and the last internal savepoint constitutes a unit of temporary persistency at the workstation. Besides internal savepoints, the state of the design transaction also represents a unit of temporary persistency at the private system.

At the private system, the designer can roll back work done by either aborting the recovery transaction or restoring R-Tr's state to a previously generated internal savepoint. On the other hand, the designer cannot directly roll back any unit of work at the public system. Since the designer cannot back out his design transaction, recovery transactions always represent the unit of durability at the private system. That is, the effects of design transactions can only be rolled back in the private database if the designer starts compensation transactions for them.

	GM1		GM2		GM3	
	server	workstation	server	workstation	server	workstation
Atomicity	Conv.-Tr	Conv.-Tr	Conv.-Tr	Conv.-Tr	Conv.-Tr	Conv.-Tr
Durability (after commit)	Conv.-Tr	Conv.-Tr	Conv.-Tr	Conv.-Tr	Conv.-Tr	Conv.-Tr
Temporary Persistency (before commit)	D-Tr's state	D-Tr's state and internal savepoints	D-Tr's state	D-Tr's state, internal and external savepoints	G-Tr's state	D-Tr's state, internal and external savepoints
work units which can be aborted by the designer	none	Conv.-Tr and the work done after internal savepoints	none	Conv.-Tr, D-Tr, and work done after internal or external savepoints	G-Tr	Conv.-Tr, D-Tr, and work done after internal or external savepoints

Note: Conv.-Tr stands for conventional transaction (e.g. S-Tr, R-Tr), D-Tr for design transaction, and G-Tr for group transaction.

Fig. 3.7: The transaction properties of the general processing models

3.6.3.2 Transaction Properties in GM2

As with GM1, conventional transactions in GM2 also represent the units of atomicity and durability at both the public and the private systems. While the design transaction is the only unit of temporary persistency at the public system, the private system must guarantee this property for D-Tr as well as for external and internal savepoints.

The designer can completely back out non-committed recovery transactions as well as his design transaction at the private system. He can also roll back work done by either restoring R-Tr's state to some internal savepoint or backing up D-Tr to a previously generated external savepoint. As with GM1, the designer cannot directly roll back work done at the public system.

3.6.3.3 Properties of GM3

As with the other general models, GM3 models the conventional transaction at both public and private systems as the unit of atomicity and durability in case of system failures. At the public system, the work unit for which temporary persistency must be guaranteed is represented by the group transaction. On the other hand, the private system must guarantee temporary persistency for D-Tr, external savepoints, and internal savepoints.

At the public system, the designer can directly back out non-committed group transactions. At the private system, he can back out recovery transactions as well as the D-Tr. He can also roll back work done by either restoring R-Tr's state to an internal savepoint or backing up D-Tr to some external savepoint.

Chapter 4

Database Recovery Requirements in the Design Environment

4.1 Recovery Situations in the Design Environment

In this section, we discuss possible failures in the design environment and investigate how these failures can affect both database consistency and database system operation. The reference system architecture derived in chapter 2 as well as the general design transaction models presented in chapter 3 constitute the framework on the basis of which we conduct the following discussion.

4.1.1 Undesired Events in the Design Environment

In [LaSt79], the set of all events which affect computing systems is divided into two mutually exclusive groups: the group of desired events and the group of undesired events. Desired events are those which make part of as well as collaborate in the correct computing activity (e.g. transactions which produce consistent database states). Failures in the computing system are examples of undesired events. The group of undesired events has been further subdivided into two categories: undesired but expected events and undesired and unexpected events. Examples of undesired but expected events are system crashes or failures in the communications network. Earthquake and nuclear war are examples of undesired and unexpected events which can damage computing systems.

At least two facts make it impossible to construct systems which cope with all kinds of undesired events. First, it is impossible to conceive all events which can occur in a system (nor all combinations of them). Secondly, it would be too expensive to protect systems against all types of failures. Usually, the number and type of undesired events the system can cope with depends on a cost-benefit compromise.

Undesired events can affect the computing system in two ways: they can produce an inconsistent system state or/and reduce system availability. Systems are more or less reliable, depending on how they can tolerate and recover from failures [MoAB86]. Mechanisms which are built to help systems tolerate and recover from failures can be installed in all layers of the computing system. Such mechanisms can be realized at the hardware, operating system, database system, and application program layers.

The recovery component of a DBS should guarantee system reliability at the database system layer. During its design, the set of failures (i.e. undesired events) it will have to cope with must be determined. This occurs on the basis of a set of requirements like the expected reliability of the overall computing system, the reliability of other system layers, the DBS architecture, and the processing model to be realized.

A system is more or less reliable depending on how well it maintains availability and correctness by executing its activities. To maintain availability at the database system layer, the DBS must guarantee that transactions can continue normal execution even in case of failures. This can be achieved only if two preconditions are satisfied. First, the DBS must be realized as a distributed database system. That is, its algorithm executes on a set of autonomous processing nodes. For this discussion, a processing node consists of a processor and a storage hierarchy associated with it. Autonomous processing nodes fail independently. That is, failures on one node do not necessarily cause other nodes to fail. The second precondition which must be satisfied by the DBS so that it guarantees system availability is related to the replication of the database in either some or even all processing nodes of the computing system. Since transactions must continue normal execution even in case of failures, they should be able to access all the data they need when either a processing node or the communications network fails.

Although the characteristics of the design environment lead to the distribution of the DBS (e.g. over a server-workstation computer configuration), the necessity and the costs of data replication in this environment are not so clear. At least until now, availability seems not to be seen as a serious problem in the design environment. The design activity is by no means a real-time application. Design transactions take long and can be suspended many times, before they terminate (i.e. usually, they consist of various design sessions). On the other hand, we can think of some situations where system availability should be preserved. For instance, the design transaction at the private system (i.e. private processing node) depends on CHECKOUT operations. Although these operations are supposed to be executed seldom, they synchronize the whole design work. Thus, it would be nice if the system could guarantee the execution of CHECKOUT operations even if the server node fails. A failure at a private node constitutes another situation where availability could be important. The designer is not able to resume his work at another workstation, if he cannot access his private database from that node. System mechanisms should be provided which permit the designer to reconstruct his private database at another processing node in case of node failure as well as in the same node in case of media failure.

While the design database system can also be realized without mechanisms which assure availability in case of system failures, it cannot dispense with the mechanisms which guarantee correctness. The correctness aspect is related to the preservation of database consistency. In standard database systems, the transaction manager is responsible for database consistency during normal processing as well as in case of failures. Since the new design processing models are extensions of the conventional transaction paradigm and, at the same time, introduce more spheres of control in the processing environment, we should investigate how these new models may influence database recovery.

We will only consider the correctness aspect of database system reliability, since this is the most important reliability aspect for the design environment (and for other environments, too). Besides, the maintenance of correctness in the design environment already represents a complex problem for which many questions have not yet been answered. These questions are related to the correctness of already existent recovery techniques in the design environment (e.g. can they guarantee serializability only on an object basis?) and the efficiency of those recovery mechanisms which can guarantee correctness in this environment (e.g. how do they influence response time in certain situations?).

To continue our investigation of database recovery requirements in the design environment, we must decide which failures of this environment the database system should cope with. To be able to investigate recovery requirements in detail, we should focus only on the critical failures which can affect normal database processing. The rest of the possible failures will be further subdivided into two categories: the failures which we assume will be handled at other system layers and the ones which will be treated as undesired and unexpected events. Before we make our assumptions, though, let us enumerate the set of undesired but expected events which will be considered here:

- *Processor Failure:* There are, at least, three ways in which the processor can fail [MoAb86]. It can simply stop working (i.e. it halts). It is also possible that the processor continues working but presents abnormal, intermittent delays. Finally, the processor can go insane. That is, it continues to work but generates incorrect results.
- *Storage Failures:* One can devise two distinct categories of database storage: volatile (e.g. main memory) and nonvolatile storage (e.g. storage space on the basis of magnetic disks or tapes). When the processor fails, the contents of volatile storage are lost. Since nonvolatile storage is usually implemented as an autonomous device, it fails independently from the processor.
- *Network Failures*: If we consider the communications network to be an autonomous hardware and software system which connects processing nodes to one another, we can assume that it fails independently from those nodes. When the network fails, messages between nodes are not passed properly anymore.
- *Network Partitions*: These failures can be caused either by a node or a network failure. The distributed database system is partitioned into groups of processing nodes. Inside each group, nodes can communicate with one another but no communication between nodes of different groups is possible.
- *Transaction Failures*: In this failure category, we group all events together which can interrupt the normal execution of some specific transaction. Transaction failures can be identified either by the transaction itself (e.g. when incorrect input data is detected), by the designer, or by the database system (e.g. by the occurrence of a deadlock involving the transaction).

4.1.2 A Failure Model for the Design Database System

Relying on the list of undesired but expected events, we make the following assumptions about a failure model for database systems in the design environment:

- We will use a fail-stop model to represent processing node activity at both the server and the workstation nodes. At any time, the node is in one of three possible states: perfect, halted, or recovery. In the perfect state, the node functions correctly. That is, the node executes its algorithm correctly and in the expected time interval. Moreover, it never stops and responds promptly to messages from other nodes. In the halted state, the node halts and does absolutely nothing else. When the node is in the recovery state, it executes a set of predefined algorithms to recover the database to a consistent state.

  As opposed to [MoAb86], we relax the requirement that the node in the recovery state may only execute recovery algorithms. Depending on the recovery technique realized, the node can start new transactions in parallel to recovery activities [KARD88]. Figure 4.1 depicts the dynamical characteristics of the fail-stop model. At the beginning of the processing activity, the node is in the perfect state. At some point in time, a node failure occurs and the node halts. Some time later, the node enters the recovery state. At this point, the database system recovers from the failure. At the end of the recovery activities, the node enters the perfect state again.

  By choosing the fail-stop model to represent processing node activity in the distributed design database system, we assume that the other two types of processor failures described above, namely correct processing with abnormal delays and incorrect processing, are treated by lower layers of the computing system.

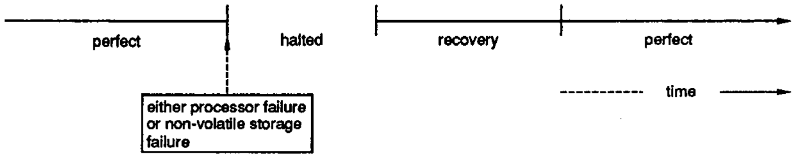


Fig. 4.1: A fail-stop model for processing nodes

- We will also represent storage failures in our failure model. The contents of volatile memory are lost every time the processor fails. On the other hand, volatile memory as well as processors remain intact when non-volatile storage fails. As with processor failures, we also consider non-volatile storage failures to be processing node failures. Therefore, we assume that the processing node immediately stops when storage failures occur.
- It will further be assumed that the hardware and operating system layer provide the database system with a reliable communications network [MoAb86]. The communications subsystem is supposed to always deliver messages in the right order and in the expected time interval. Furthermore, we assume that this system neither duplicates messages nor generates spontaneous messages. Finally, processing nodes interpret network failures (e.g. partition, break down, loss of messages) as failures at the nodes with which they want to communicate.
- All transaction failures are handled by the recovery component of the database system. Processing situations which can require recovery activity in the scope of transactions are deadlock situations as well as blocking situations involving CHECKOUT operations, user-requested transaction backout, and restoring the transaction state to a previously generated savepoint. According to the general processing models of chapter 3, deadlock situations can take place at the public and the private systems. The same holds for blocking situations. On the other hand, we assume the designer can only abort transactions that run at the workstation. Consequently, transactions being executed at the public system may also have to be aborted. Finally, the generation as well as restoration of savepoints always occur at the private system (with possible consequences at the public system, too).

4.2 Recovery Protocols for the Design Database System

In this section, we present a set of integrated recovery and communication protocols which cope with the failures derived in the last section. These protocols assume a one-node failure model. That is, they suppose that processing nodes fail independently and not at the same time. Moreover, it is assumed that a node does not fail while another one is recoverying from a failure. On the other hand, the protocols support the situation where a node fails while recoverying from a previous failure.

The recovery protocols rely on properties of the general processing models introduced in chapter 3. Moreover, they consist of basic recovery and communication actions and describe the database recovery activity in a general form. Therefore, this set of protocols represents no specific implementation strategy for database recovery in the design environment. In principle, any recovery technique intended to guarantee database recovery in this environment should be based on these protocols, though. The protocols describe recovery actions that cope with different recovery situations, respectively. Since these algorithms cover recovery activities for all general processing models of chapter 3, we believe that optimizations by specific implementations are possible.

4.2.1 Further Assumptions about the Design Database System

Besides the assumptions made so far, the recovery and communication protocols to be presented here have been designed under the following additional suppositions:

- The DBS controls the execution of (sub)transactions at remote processing nodes (e.g. server node) by enforcing an extended version of the centralized two-phase commit protocol discussed in [MoAb86].

  Recovery transactions running at the workstation can start remote operations at the server node asynchronously. That is, the designer or the private system at the workstation need not wait for the results of a remote operation before they can start another remote operation at the server node.

  To start a remote operation at the server, the private system sends a corresponding **start-work** message and resumes its own processing work (i.e. the recovery transaction at the workstation). By receiving a **start-work** message, the public system begins executing the required task (i.e. it starts a short transaction to execute the request). If the work at the server node terminates successfully, the public system saves its results (i.e. the short transaction enters the prepared state) and sends the private system an **ok** message. Otherwise, the public system aborts the short transaction and sends a **nok** message to the private system.

  If the recovery transaction terminates successfully, the private system saves its results and waits for the messages of the public system. Note that, for each remote operation started at the server node, the private system waits for one reply (either **ok** or **nok**) of the public system.

  In case the recovery transaction must be backed out at the workstation, the private system sends the public system one **abort** message for each remote operation started at the server node. When the public system receives an **abort** message, it immediately rolls back the corresponding short transaction (no matter the state of it).

  If the private system receives **ok** messages for all remote operations and the recovery transaction terminates successfully, it sends the public system one **commit** message for each remote operation started. When the public system receives a **commit** message it commits the corresponding short transaction, if this transaction is in the prepared state.

  If the recovery transaction terminates successfully but the private system receives some **nok** messages from the public system, it (or the designer) can decide either to abort the recovery transaction and all successfully terminated short transactions, or to restart those remote operations which failed, or even to start new (alternative) remote operations. In contrast to the centralized two-phase commit protocol discussed in [MoAb86], these three alternatives are possible in our extension, because the private system always sends the public system one transaction termination reply (either **commit** or **abort**) for each remote operation started at the server node.

  We explain our extension to the two-phase commit protocol in [MoAb86] with an example which is illustrated in Figure 4.2. Suppose the designer develops his design by means of a design tool. The whole execution of such a graphic program (including remote operations) will be realized as one (possibly long) recovery transaction at the private system. Suppose further that the design being developed relies on objects O_1 and O_2. First, the private system starts a CHECKOUT operation at the public system to copy O_1 into the private database. After O_1 is sent to the workstation and the server has saved its related CHECKOUT locks in a local log file, the public system sends an **ok** message to the private system and the design tool begins processing O_1 at the workstation. Note that, at this point in time, the server has not yet committed the short transaction which executes the CHECKOUT operation related to O_1 at the server node. Before checking out O_2, the design tool generates an internal savepoint (isvp) for the recovery transaction. If the CHECKOUT (sub)transaction which should copy O_2 fails at the public system, the design tool can decide either to back out the recovery transaction completely, or to restore R-Tr´s state to the previously generated savepoint and try to copy O_2 again (or start another CHECKOUT operation to copy, for instance, object O_3). The second alternative helps preserving the work already done on the basis of O_1.

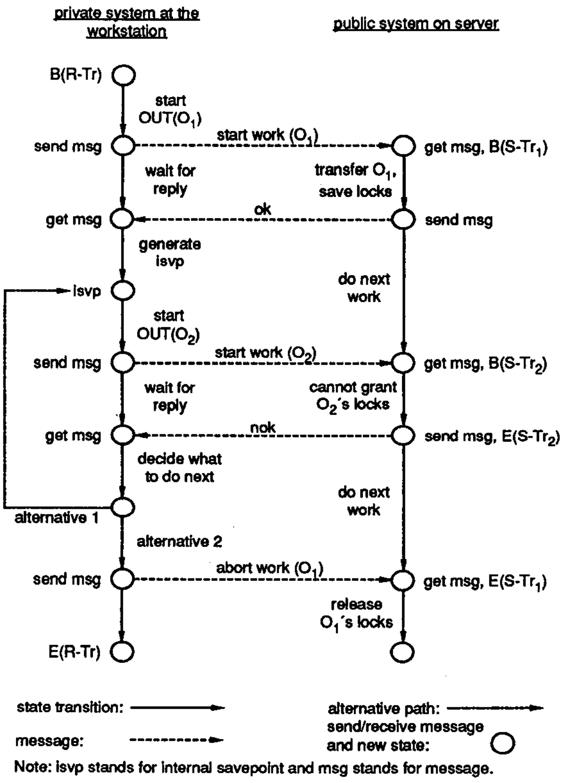


Fig. 4.2: Extending the commit protocol to cope with savepoints

- Remote transactions which execute CHECKOUT operations at the public system cannot be blocked forever. If the desired object has already been locked by another transaction in an incompatible mode, the remote (sub)transaction checking out this object is aborted by the public system and a **nok** message (possibly together with some explanation) is sent to the private system. The designer, then, must decide what to do next.
- CHECKIN operations are executed by special recovery transactions (i.e. CHECKIN recovery transactions). These transactions execute no other operations. On the basis of this assumption, we can simplify the recovery protocols. This simplification leads to no loss of generality by the protocols, though.
- Remote (sub)transactions can be executed asynchronously. That is, remote operations which are started by the same recovery transaction can be executed in parallel at the server node.

- In case the recovery protocols are applied to an environment which realizes the GM3 design processing model, the public system will always request design transactions to abort, if they try to copy objects either from or into the group database of an inexistent (e.g. aborted) group transaction.
- We assume the transaction manager of the database system associates a state record with every transaction which is started in the design environment. Moreover, every data system related to the execution of a transaction (e.g. the public system) keeps its own version of the transaction´s state record. Thus, every system involved in the transaction execution can keep track of exactly those informations about the transaction it needs.

  In systems which realize either GM1 or GM2, we assume that the state record kept by the public system for a design transaction contains, at least, information about the objects the transaction checked out (e.g. object identifier, lock mode, version number, if the object has already been checked back into the public database).

  At the private system, the state record of the design transaction is assumed to store the list of committed short update operations which have been started from inside of it as well as information about external savepoints. We also suppose that the private system keeps state records for running recovery transactions, too. These records mainly store information about internal savepoints as well as remote operations which have been started from inside these transactions.

  In systems which realizes the GM3 processing model, we suppose the public system associates one state record with every running group transaction. Every G-Tr state record contains information about the state of the respective design (sub)transactions at the public system as well as the objects and locks of the respective group database.

4.2.2 The Basic Actions of the Recovery Protocols

In the following, we introduce a set of processing primitives which form the basis of our integrated recovery and communication protocols. Later on in this section, a specific sequence of processing primitive calls will be associated with every recovery situation. The first three primitives to be presented are based on the recovery actions proposed in [HäRe83].

- **Undo** *(tr):* Partial-UNDO action described in [HäRe83]; it rolls back the work unit *tr.*
- **Gundo** *(set):* Global-UNDO action; all work units which are elements of *set* are backed out; *set* imposes a partial order on its elements.
- **Redo** *(set):* Combines the effects of the Partial-REDO and Global-REDO actions. That is, the results of all work units which are elements of *set* will be restored in the database. *set* imposes a partial order on its elements.
- **Restore** *(transaction state):* Restores the state record of some transaction. At the same time, this primitive can use the data kept by this record to restore other system informations (e.g. public system´s lock table), too.
- **Delete** *(data structure):* Deletes some data structure of the system (e.g. private database) and releases storage space related to it.
- **Restart** *(tr):* Restarts the execution of the work unit *tr.*
- **Decide** *(first option* **or** *second option):* This primitive represents a decision to be taken by the designer. Either the first option or the second option will be executed.
- **Inform** *(information):* Shows some information to the designer (e.g. the list of all committed short update operations which have been started from inside the design transaction).
- **Msg** *(receiver,msg,explanation):* Sends the processing node identified by *receiver* the message (i.e. character string) represented by *msg* and, possibly, some explanation about the reasons for sending the message (*explanation*). In the following, we use the

character "&" to represent the string concatenation operator inside the message processing primitive. Further, we consider all operands of & to be character strings.

- **Receive** *(msg):* Presents the processing node the message *msg* which has been sent to this node by means of a **Msg** statement
- **Wait** *(wait-for clause):* Represents a waiting time for the recovery mechanism. The *wait-for clause* indicates what the recovery should wait for.

4.2.3 Some Procedures to Simplify the Description of the Protocols

Since some specific sequences of processing actions are executed by various of the recovery protocols to be presented, we decided to describe them here as procedures that can be called by the protocol programs. These procedures as well as the protocols themselves are described by means of a Pascal-like programming language. Note that upper-case characters are used to begin new statements. Moreover, comments on the code can be written using either the "Comment" statement or inserting them between a "/*" string and a "*/" string.

- PROCEDURE ACTION1 (R-Tr):

```
Comment: Action1 is executed by the private system at the workstation. If
         R-Tr has previously generated at least one internal savepoint, this
         procedure restores R-Tr´s state to its youngest savepoint.
         Otherwise, R-Tr is completely backed out;
Begin
If R-Tr has at least one internal savepoint
   then Restore (R-Tr´s state to its youngest savepoint)
   else Undo (R-Tr);
remoteop := Select those R-Tr´s remote operations which should be undone;
For every remote operation OPi in remoteop do
   Msg (server node, 'abort-operation'& OPi´s identifier);
Inform (list of undone operations, cause: abort-or-restore savepoint);
End; /* action1 */
```

- PROCEDURE ACTION2 (private system´s messages):

```
Comment: Action2 is executed by the public system at the server node. By
         receiving a set of abort messages from the private system at the
         workstation, the public system aborts the corresponding S-Trs.
         Every message received contains two informations. First, it
         indicates which operation should be executed (i.e. the abort
         operation, in this case). Secondly, the message contains the
         identifier of the remote operation associated with the S-Tr which
         should be aborted;
Begin
While Receive (msg = 'abort-operation') do
   Undo (the S-Tr associated with OPi´s identifier);
End; /* action2 */
```

- PROCEDURE ACTION3:

```
Comment: Action3 is processed by the private system at the workstation to
         back out all running R-Tr. This procedure is called in three
         situations: by rolling back D-Tr to an external savepoint, by
         completely aborting D-Tr, and by aborting a group transaction;
Begin
For every running R-Tr do
   Begin
   Undo (R-Tr);
   remoteop := Select all remote operations which have been started by R-Tr;
   For every operation OPi in remoteop do
      Msg (server node, 'abort operation'& OPi´s identifier);
   End; /* for */
End; /* action3 */
```

- PROCEDURE ACTION4 (ctr, aborting, public-or-group-database):

```
Comment: Action4 is executed by the private system to either CHECKIN or
         UNCHECKOUT objects in two possible situations: by backing
         up D-Tr´s state to an external savepoint or by aborting D-Tr. The
         parameters aborting and public-or-group-database are both
         boolean. ctr is a set of committed transactions;
Begin
outset := Select all objects which have been checked out by some R-Tr in ctr;
For every object Oi in outset do
   If aborting
      then Msg (server node,'UNCHECKOUT'& Oi´s identifier);
         /* check Oi into the public database without changes -> UNCHECKOUT */
      else  Begin /* restoring D-Tr´s state to an external savepoint */
            Inform (outset);
            Decide (either check Oi in the public database without changes
                    or Maintain Oi in the private database);
            End; /* else */
shortupd := Select all committed remote update operations which have been started by
            some R-Tr in ctr;
Inform (shortupd);
For every operation OPi in shortupd do
   Decide (either Start a compensation operation for OPi or Do-Nothing);
Wait (for all necessary responses of the server node);
End; /*action4 */
```

- PROCEDURE ACTION5:

```
Comment: Action5 is executed by the public system at the server node when
         a D-Tr is either backed up to an external savepoint or completely
         backed out by the designer at the workstation;
Begin
While Receive (msg = 'UNCHECKOUT') do
   Release CHECKOUT locks for object with Oi´s identifier in the public database;
End; /* action5 */
```

- PROCEDURE ACTION6 (remote operation´s identifier):

```
Comment: Action6 may have to be executed by the private system at the
         workstation whenever either a deadlock or a blocking situation
         occurs at the server node;
Begin
Identify the R-Tr related to the remote operation´s identifier;
Inform (long-duration blocking or deadlock at the public system);
Decide (either Undo (R-Tr) or Restore (R-Tr´s state to an internal savepoint));
End; /* action6 */
```

- PROCEDURE ACTION7 (inset):

```
Comment: Action7 is executed by the public system at the server node when
         a D-Tr is either backed up to an external savepoint or it is aborted,
         and the DBS realizes the GM3 processing model; inset
         represents a set of data objects which have been checked back
         into the database by some transaction.
Begin
For every object $O_i$ in inset do
   Begin
   transset := Select all other design transactions which have checked $O_i$ out of the
               group database after D-Tr had updated it;
   For every transaction $T_k$ in transset do
      Case $T_k$.state of
      Begin
         suspended:   Append (msg = 'invalidated object'& $O_i$´s identifier) to $T_k$´s
                      state record at the public system;
         aborted:     Do-Nothing;
         active:      Msg ($T_k$´s node,'invalidated object'& $O_i$´s identifier);
         ready:       Begin
                      $T_k$´s state := active;
                      Inform the design administrator ($T_k$ is active again, $O_i$ has
                                                        been invalidated);
                      End; /* ready */
      End; /* case */
   End; /* for every $O_i$ */
End; /* action7 */
```

4.2.4 Recovery Protocols Based on Transaction Serializability

After describing the main operations to be executed by the recovery protocols as well as defining the most frequently executed sequences of these operations in the form of procedures, we are ready to present the protocols themselves. In this section, we introduce those integrated recovery and communication protocols which are based on and support serializable transaction schedules. Algorithms of this group guarantee the maintenance of transaction serializability in case of failures. In the next section, we will present those recovery protocols which are based on object-oriented two-phase locking protocols.

On the basis of our failure model, we can specify for each one of the general processing models proposed in chapter 3 the recovery protocols which are necessary and sufficient to guarantee database consistency in case of failures. If the DBS realizes GM1, the recovery manager must be capable of executing the following tasks:

- Transaction undo for S-Tr at the server node and R-Tr at the workstation.
- Restore the state of R-Tr to some previously generated internal savepoint.
- Transaction redo for D-Tr at the server and at the workstation, S-Tr redo at the server, and R-Tr redo at the workstation.

Since under GM2 D-Tr is atomic at the server node, the recovery manager should also support D-Tr backout in this environment besides realizing the recovery actions listed above. Moreover, in the GM2 environment the recovery manager must also support the generation and restoration of external savepoints for D-Tr.

In the GM3 environment, the recovery manager must realize all the activities listed above, too. It must take into consideration, though, that the operations to abort D-Tr as well restore D-Tr´s state to an external savepoint must be based on object-oriented serializability. Besides, the GM3 environment requires that the recovery manager be able to restore G-Tr´s state in case of a system crash at the server node and abort G-Tr on user request.

In the following, we present a set of ten recovery protocols. They respectively recover the state of the database system in the following situations: node failure at the public system, node failure at the private system, deadlock situation at the public system, deadlock situation at the private system, blocking situation at the public system, backing up R-Tr´s state to an internal savepoint, user-requested R-Tr backout, backing up D-Tr´s state to an external savepoint, user-requested D-Tr backout, and user-requested G-Tr backout.

The protocol to back up D-Tr to some external savepoint as well as the one which aborts D-Tr are only valid for systems which realize the GM2 processing model. In the next subsection, we introduce two equivalent protocols which rely on object-oriented two-phase locking to respectively restore D-Tr´s state to some external savepoint and back out D-Tr in the scope of the group transaction (i.e. GM3).

In the following, we will not present algorithms for data saving activities. Depending on the recovery technique to be realized, data saving activities can be very different (e.g. compare these activities by log techniques and by shadow techniques). We suppose that enough information is saved during normal system execution so that the recovery protocols can be correctly executed in case of failures.

4.2.4.1 Recovery by Node Failure at the Public System´s Site

- Public system´s actions:

```
Comment: After node restart, the public system identifies the transactions
         which were running or had already committed before the crash.
         Corresponding messages are sent to the respective private
         systems. Besides that, the public system executes undo as well as
         redo actions on behalf of those transactions;
- ctr := Select the group of committed S-Tr;
- ptr := Select the group of prepared S-Tr;
    /* non-committed transactions which already saved results */
- utr := Select the group of unsaved S-Tr;
    /* transactions which are neither in ctr nor in ptr */
- Gundo (utr);
- Redo (ctr U ptr);
- Case transaction model of
    Begin
    GM1: GM2: Restore (the state records of all running D-Tr);
    GM3: Restore (the state records of all running G-Tr);
    end; /* case */
- For every S-Tr in utr do
    Msg (user node,'nok'& identifier of the remote operation related to S-Tr,failure);
- For every S-Tr in ptr do
    Msg (user node,'ok'& identifier of the remote operation related to S-Tr,failure).
```

- Reaction at the private system:

```
- If (msg = 'ok')
    then if the R-Tr related to the corresponding remote operation is committed
        then Msg (server node,'commit operation'& remote operation´s identifier)
        else Do nothing
    else if (msg = 'nok')
        then ACTION1 (R-Tr related to the corresponding remote operation).
```

4.2.4.2 Recovery by Node Failure at the Private System´s Site

Comment: When recoverying from a node failure, the private system analyzes those transactions which were running or had already terminated by the time the crash occurred. Committed R-Trs are redone, while interrupted ones which have a savepoint are backed up to the youngest savepoint. Both running transactions without savepoint and aborted transactions are completely backed out.

- Private system´s actions:

```
- ctr := Select the group of committed R-Tr;
- str:= Select the group of R-Tr which are not in ctr but have at least one
        internal savepoint;
- utr := Select the group of R-Tr which are neither in ctr nor in str;
- For every R-Tr in (str U utr) do
    ACTION1 (R-Tr);
- Redo (ctr);
- Restore (D-Tr´s state record).
```

- Reaction at the server node:

```
- ACTION2 (ACTION1´s messages).
```

4.2.4.3 Deadlock Situation at the Public System

- Public system´s actions:

```
-.Select a short transaction to be aborted (victim S-Tr);
- Undo (victim S-Tr);
- If only short-duration locks are involved in the deadlock situation
    then Restart (victim S-Tr)
    else Msg (user node,'nok'& remote operation´s identifier,deadlock).
```

- Reaction at the private system:
  - ACTION6 (the remote operation´s identifier sent by the public system).

4.2.4.4 Deadlock Situation at the Private System

Comment: Since it is possible that recovery transactions execute concurrently at the private system, deadlocks can take place at the workstation, too. In case of a deadlock, the private system simply selects a R-Tr to be aborted, rolls it back, and inform the public system to abort those remote operations started from inside the victim R-Tr;

- Private system´s action:
  - Select a victim R-Tr to be aborted;
  - ACTION1 (victim R-Tr).

- Reaction at the public system:
  - ACTION2 (ACTION1´s messages).

4.2.4.5 Blocking Situation at the Public System

- Public system´s actions:

```
- If the S-Tr to be blocked waits for short-duration locks /* no CHECKOUT locks */
  then block (S-Tr) /* independent of the operation it executes */
  else  begin
        Undo (S-Tr);
        Msg (user node,'nok'& identifier of the remote operation related to S-Tr,
             long-duration blocking);
        end.
```

- Reaction at the private system:
  - ACTION6 (remote operation´s identifier sent by the public system).

4.2.4.6 Backing Up R-Tr to an Internal Savepoint

- Private system´s actions:

```
- Restore (R-Tr´s state to a previously defined internal savepoint);
- For every remote operation OPi which must be undone do
    /* those which have been started from R-Tr after the generation of the savepoint */
    Msg (server node,'abort-operation'& OPi´s identifier).
```

- Reaction at the server node:
  - ACTION2 (private system´s messages).

4.2.4.7 User-Requested R-Tr Backout

- Private system´s actions:
  - Undo (R-Tr);
  - remoteop := Select all remote operations which have been started by R-Tr;

- For every operation OP_i in remoteop do
  Msg (server node,'abort-operation'& OP_i´s identifier);
- Inform (list of undone operations).

- Reaction at the server node:
  - ACTION2 (private system´s messages).

4.2.4.8 Backing Up D-Tr to an External Savepoint (Only valid for GM2)

Comment: When restoring the state of D-Tr to a previously defined external savepoint, the private system must abort all currently running R-Trs, roll back the effects of all committed R-Trs which executed at the workstation after the generation of the savepoint, and ask the server node to abort all non-committed remote operations which were started after the savepoint generation. Furthermore, the private system must inform the designer about those committed remote operations which have been started at the server node by recovery transactions which must be undone. The designer, then, must decide, if compensation operations must be started for those remote operations;

- Private system´s actions:
  - ACTION3;
  - ctr := Select all committed R-Tr which have been started after savepoint generation;
  - Gundo (ctr in the private database);

  ACTION4 (ctr, aborting := false, public-database);

- Reaction at the server node:
  - ACTION2 (ACTION3´s messages);
  - ACTION5 (ACTION4´s messages).

4.2.4.9 User-Requested D-Tr Backout (Only valid for GM2)

- Private system´s actions:
  - ACTION3;
  - ctr := Select all committed R-Tr which have been started from inside of D-Tr since D-Tr begin;

  ACTION4 (ctr, aborting := true, public-database);
  - Msg (server node,'D-Tr abort'& D-Tr´s identifier);
  - Delete (private database).

- Reaction at the server node:
  - ACTION2 (ACTION3´s messages);
  - ACTION5 (ACTION4´s messages);
  - Receive (msg = 'D-Tr abort');
  - D-Tr (D-Tr´s identifier).state := aborted.

4.2.4.10 User-Requested G-Tr Backout (Only valid for GM3)

Comment: When aborting G-Tr, the public system at the server node must first request all yet running design transactions which are descendant of G-Tr to abort. Then, the public system releases the locks held by G-Tr in the public database and deletes the group database on server.

- Public system´s actions:
  - For all running G-Tr´s design (sub)transactions do
    Msg (user node,'abort-D-Tr'& D-Tr´s identifier,G-Tr-abort);
  - Release G-Tr´s locks in the public database;
  - Delete (group database);
    /* if the group database is realized as a part of the public database, then */
    /* restore the state this part was in when G-Tr began */
  - Inform design administrator (G-Tr has been aborted);
  - Receive and Execute all requests made by private systems processing related D-Trs.

- Reaction at private nodes which run G-Tr´s design (sub)transactions:
  - ACTION3;
  - shortupd := Select all committed short update operations started from inside D-Tr;
  - Inform (shortupd);
  - For every operation OP_i in shortupd do
    Decide (either Start a compensation operation for OP_i or Do-Nothing);
  - Wait (for all necessary responses of the server node);
  - Delete (private database);
  - Inform (D-Tr aborted).

4.2.5 Recovery Protocols Based on Object-Oriented Two-Phase Locking

In our study, database recovery in systems which realize the GM3 processing model must support D-Tr backout and external savepoints in an environment where database consistency is preserved on the basis of a object-oriented two-phase lock protocol. The recovery mechanism must, therefore, be able to cope with cascading aborts.

To better understand and follow the design activities of the group transaction in GM3, we can represent its dynamical characteristics using a directed graph G. Every design (sub)transaction of G-Tr is represented as a node of G. Design transactions are inserted in G when they are created. A D-Tr is removed from G either when it commits or aborts.

The edges of G respectively represent sets of object exchange activities in G-Tr. Each object exchange activity can be expressed as a triple of the form (D-Tr_x,D-Tr_y,O_i^v) where D-Tr_x identifies the transaction which created the new version V of object O_i and D-Tr_y identifies one of the transactions which checked O_i^v out of the group database, after it had been checked back in there by D-Tr_x.

The edge E(x,y) of G represents the set of all object exchange activities between D-Tr_x and D-Tr_y which have the former transaction as their origin. Therefore, every edge of G can be expressed as a triple of the form (x,y,s_{xy}) where s_{xy} is the set of all object versions which have been created by D-Tr_x and checked out by D-Tr_y. Note that the term object version is used here to identify specific object states. Contrary to TM5, GM3 does not require that G-Tr be realized on the basis of an object version mechanism.

We explain the directed graph G with the following example which is illustrated in Figure 4.3. For this example, we assume that object updates are represented as new object versions in the group database. Suppose G-Tr consists of the set $D=\{T_1,T_2,T_3,T_4,T_5\}$ of design (sub)transactions and the group database GDB. At G-Tr begin, the state of GDB is expressed by the object set $OS=\{O_1{}^1,O_2{}^1,O_3{}^1\}$. T_1 checks $O_1{}^1$ out of GDB, creates a new version of it (i.e. $O_1{}^2$), and checks it back into GDB. Meanwhile, T_2 checks $O_2{}^1$ out, updates it, and checks $O_2{}^2$ into GDB. T_3 copies $O_1{}^2$, $O_2{}^2$, and $O_3{}^1$ into its private database. It creates the new object versions $O_1{}^3$ and $O_3{}^2$ and checks them into GDB, too. T_4 checks $O_1{}^3$ out of GDB and processes it. T_5 copies $O_3{}^2$ into its private database and creates the object version $O_3{}^3$. After being checked into the group database, $O_3{}^3$ is checked out by T_2.

As GM3 was derived in chapter 3, we explained that design transaction synchronization follows the so-called predicatewise two-phase protocol of [KoKB87] and supposed that the relationship between the objects of the database and the conjuncts of the consistency constraint associated with it represents a bijective function. A transaction can, therefore, check a new version of an object into the group database as soon as this version preserves the conjunct related to the object being updated. Since transactions are non-two-phase, they can execute CHECKIN operations at any time.

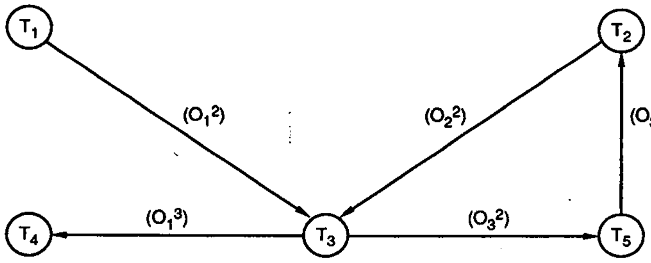

$$\begin{tikzcd}
T_1 \arrow[dr, "(O_1{}^2)"] & & & T_2 \arrow[dll, "(O_2{}^2)"'] \\
T_4 & T_3 \arrow[l, "(O_1{}^3)"'] \arrow[rr, "(O_3{}^2)"] & & T_5 \arrow[u, "(O_3{}^3)"']
\end{tikzcd}$$

Fig. 4.3: Representing G-Tr as a directed graph

There are, at least, two situations where the designer may want to either back out the design transaction or back it up to some external savepoint: he can either realize that his design is not logically correct (with respect to the application) or conclude that some new object version that he checked in does not really preserve the consistency constraint (it is possible that not all predicates can be automatically tested by the system).

It is possible that other design transactions get involved in the process of rolling back work of a specific design transaction. If, for instance, T_1 must be aborted, then T_3 and possibly T_4 will have part of their works rolled back, too. T_1 backout already represents a complex task when all involved transactions are active. How can it be done, if some of these transactions have already committed? We solve this problem by introducing a new state for design (sub)transactions in G-Tr, namely the **ready state**.

A design transaction always is in exactly one of six states: **non-existent** (i.e. D-Tr has not yet been created), **active**, **suspended** (i.e. D-Tr has been temporarily deactivated by the designer), **ready** (i.e. the designer has notified the system of his intention to commit D-Tr), **aborted**, or **committed**. Figure 4.4 shows the state transition diagram for design transactions in GM3.

Instead of directly committing his design transaction T, the designer transfers to the system the control over it by executing the READY-TO-COMMIT statement. The system brings the transaction into the ready state by checking all its updated objects for which no CHECKIN operation have been executed back into the public database.

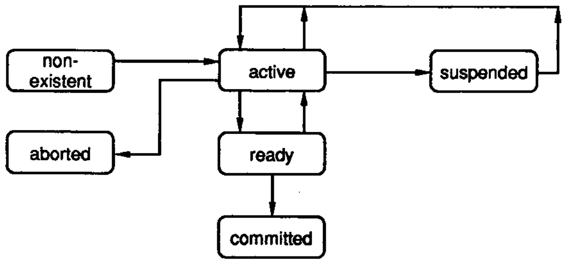


Fig. 4.4: State transition graph for D-Tr in GM3

T's transition into the committed state depends on the state of other design transactions. We suppose the public system maintains a directed graph G for G-Tr (as part of the state record of this transaction). By analyzing G, the public system can decide the next state transition of each ready design transaction in G-Tr. If any of the transactions from which T saw results aborts, T must be brought back into the active state and the designer must be informed that some of the object versions he used in his design are not valid anymore. The designer, then, can decide either to continue executing T on the basis of other object versions or to abort it. This same procedure might also have to be executed for T if some transaction from which T saw results backs up to a previously generated external savepoint. While conventional recovery techniques must resolve conflicts as the ones described above by forcing cascading aborts in the group transaction environment, recovery based on the G graph supports partial rollback of involved transactions. By allowing ready design transactions to be reactivated, the G graph can prevent significant parts of the design work from being backed out in case of failures in the group transaction environment.

A design transaction can enter the committed state only when all its ancestor transactions in G have either committed or are able to do that. Let us take the commit process of T_3 in Figure 4.3 as an example. When the designer terminates his design work, he declares T_3 to be in the ready state. T_3 directly depends on T_1 and T_2 to commit. Indirectly, T_3 depends on both T_5 (and on itself) to commit.

Let us first consider the case where all ancestors of T_3 enter the ready state. By entering the ready state, T_1 can automatically commit because it has no ancestors. If T_1 commits, it is removed from G and will not be further considered in the commit process of T_3. Since T_2 saw some updates of T_5, it can commit only if T_5 commits. The latter depends on T_3 to commit, though. This constitutes a cycle in G. Cycles of ready design transactions in G can be broken only when all ancestors of all nodes in the cycle achieve the ready state.

Figure 4.5 shows a possible expansion of the graph shown Figure 4.3. Looking at this expansion, it is easy to understand why not only the nodes involved in the cycle but also their ancestors must be in the ready state for T_3 to commit. If either T_6 or T_7 aborts or backs up to some external savepoint, T_5 might have to be brought back into the active state. This might cause T_2 to enter the active state, too. If, as a consequence, T_2 decides to abort, T_3 will have to be reactivated (i.e. will not be committed).

If any ancestor of T_3 aborts, the public system must analyze all paths that connect the aborting transaction with T_3 to decide if this transaction must be reactivated or not. If T_3

has checked out any of the objects processed by the aborting transaction, it must immediately re-enter the active state. Otherwise, the reactivation of T_3 will depend on the final state of ancestor transactions that re-entered the active state.

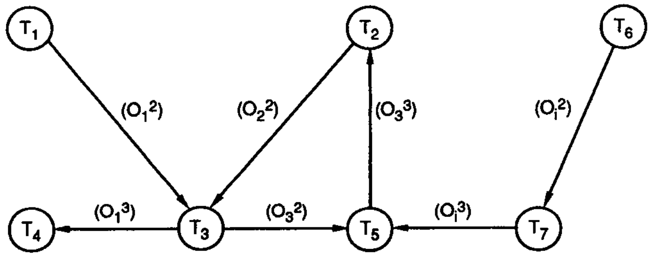


Fig. 4.5: A possible extension of the graph shown in Figure 4.3

Now let us go back to Figure 4.3 and discuss the situation where the designer backs up T_3 to some external savepoint ESVP. Suppose ESVP has been generated at some time between $O_3{}^2$'s CHECKIN and $O_1{}^3$'s CHECKIN. Thus, by restoring the state of T_3 to ESVP, $O_1{}^3$ must be invalidated and all transactions which have checked it out of the group database must roll back the work done on its basis.

In the directed graph of Figure 4.3, only T_4 has checked $O_1{}^3$ out of GDB. If T_4 is in the aborted state, its work has been rolled back already. If it is active, its owner (i.e. the designer) must be informed that $O_1{}^3$ became an invalid object version. If T_4 is suspended, the system must somehow wait until the designer reactivates it to inform him of what happened. Finally, if T_4 is ready to commit, the system must return it to the active state and inform either its owner or the overall design administrator that $O_1{}^3$ has been invalidated. From what has been explained above, it is easy to understand that T_4 could not be in the committed state by the time T_3 is backed up to ESVP. Since the system synchronizes design transaction work on an object basis, descendant transactions which have checked out objects which had been released in GDB, before the corresponding transaction has issued an external savepoint, need not be bothered by the respective RESTORE operation. Thus, neither T_5 nor T_2 gets involved in the restoration of ESVP.

There exist some similarities between the **prepared** state of the two-phase commit protocol and the **ready state** associated with the G graph. Both represent transaction states where the transaction has already concluded its work but must wait for external events to really commit it. The ready state differs from the prepared state, though, in that from the latter the transaction can only commit or abort its work, while from the former the transaction can also go back into the active state.

There are yet other differences between the two-phase commit protocol for distributed transactions and the commit protocol proposed for GM3. The two-phase protocol places the decision to commit the whole transaction hierarchy on the coordinator transaction and forces all transactions to commit at the "same time" (i.e. when the immediate ancestor terminates). Furthermore, this protocol does not permit that transactions have more than one immediate ancestor (i.e. transaction hierarchies always build a tree-like dependency graph). The commit protocol proposed here for design transactions in GM3 uses no coordinator, allows transactions to have more than one immediate ancestor, and permits (sub)transactions to commit at different points in time.

The transaction cooperation concepts proposed for GM3 also differ from the traditional nested transaction concept presented in [Moss81]. Similar to normal distributed transactions, nested transactions can build only tree-like hierarchies. That is, every sub-transaction can have only one parent (immediate ancestor), from which it can see temporary results (by inheriting the parent locks). Further, the traditional nested transaction concept does not allow ancestor transactions to see temporary results of their own sub-transactions. In [HäRo87], nested transaction environments which permit that ancestors see temporary results of their children (sub-transactions) have been investigated but no complete protocol has been proposed to handle these environments.

In [PROF85], the architecture of a distributed operating system is presented that integrates the nested transaction concept with a modified version of the two-phase commit protocol for distributed transactions. Figure 4.6 shows the state diagram for transactions in PROFEMO. In the **active** state, the (sub)transaction executes its algorithm normally. At the end of its work, the transaction enters the **completed** state. In this state, locks can be released and inherited to the ancestor transaction which, then, can see the sub-transaction results as well as allow other sub-transactions to further process them.

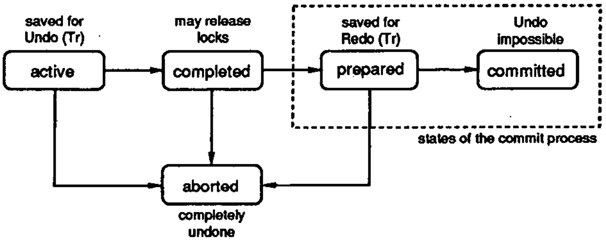


Fig. 4.6: PROFEMO's transition state diagram

In PROFEMO, users can decide which transactions of the nested hierarchy should participate in the committing process. Those completed sub-transactions from which ancestors have already seen results need not be involved in the expensive commit process. By committing an ancestor which has already acquired the locks of its completed sub-transactions, PROFEMO actually commits those sub-transactions, too.

The nesting of transactions as well as commit and abort processes in the distributed environment are controlled by means of a so-called recovery graph in PROFEMO. This graph is based on the ideas presented in [Davi78] and [Rand78]. Although the recovery graph keeps track of transaction dependencies much in the same way the G graph proposed to represent the group transaction does, the former has always a tree-like structure, since every sub-transaction in PROFEMO can acquire locks by only one ancestor transaction. The recovery graph does not keep track of the specific objects being exchanged by transactions as the G graph does, because sub-transactions in PROFEMO must always backout if their ancestor aborts.

The **completed** transition state proposed in [PROF85] cannot be compared with the **ready state** proposed for design transactions in GM3. First, although transactions in the completed state can already release locks held, their results have not yet been saved by the system. Secondly, from the completed state transactions cannot return to the active state anymore. They must enter either the prepared or the aborted state.

Actually, the nested transaction theory cannot cope well with the group transaction concept because design transactions can also be only partially dependent from each other. That is, design transactions may exchange semi-committed results without having to establish a parent-child relationship for that. This "partial dependency" of design transactions is expressed by the ready state introduced here and the possibility of transactions which are in the ready state to reenter the active state.

After having described how transaction cooperation in GM3 can be realized, we can introduce the recovery protocols to restore the state of D-Tr to some external savepoint and to abort D-Tr in GM3. Both protocols are based on object-oriented two-phase locking and have been designed for the case transaction cooperation in GM3 is implemented as described above.

4.2.5.1 Backing Up D-Tr to an External Savepoint (Only valid for GM3)

Comment: Although this algorithm pursues the same goals as the one presented in paragraph 4.2.4.8, it does it in an environment where D-Tr updates which must be rolled back may have been seen by other design transactions, already;

- Private system´s actions:

```
- ACTION3;
- ctr := Select all R-Tr which have committed after the savepoint generation;
- Gundo (ctr in the private database);
- ACTION4 (ctr, aborting := false, group-database);
- Msg (server node,'esvp'& D-Tr´s identifier & esvp´s identifier).
/* esvp stands for external savepoint */
```

- Reaction at the server node:

```
- While receive Msg (receiver,msg,explanation) do
    Case msg of
    Begin
      abort-operation:       Undo (corresponding S-Tr);
      UNCHECKOUT(O_i):       Release CHECKOUT locks for O_i in the group
                             database;
      start-short-update:    Start a S-Tr to execute a short update operation;
      esvp(D-Tr-id,esvp-id): Begin    /* the term id stands for identifier */
                             inset := Select all objects which have been checked in
                                      by D-Tr, after the savepoint generation;
                             ACTION7 (inset);
                             End; /* generate external savepoint */
    End. /* case msg */
```

- Reaction at nodes of affected design transactions:

```
- Decide (either Restore (D-Tr´s state to some external savepoint)
          or Decide (either Back-Out D-Tr or Continue D-Tr)).
```

4.2.5.2 User-Requested D-Tr Backout (Only valid for GM3)

Comment: Since D-Tr updates may have been seen by other design transactions belonging to the same G-Tr, this algorithm must rely on the group transaction´s graph G to abort D-Tr; While the private system rolls back D-Tr at the workstation, the public system analyzes G to identify other design transactions which should be informed that D-Tr is being aborted;

- Private system´s actions:

```
- ACTION3;
- ctr := Select all committed R-Tr which had been started from inside D-Tr;
```

- ACTION4 (ctr, aborting := true, group-database);
- Delete (private database);
- Msg (server node,'D-Tr-backout'& D-Tr´s identifier).

- Reaction at the server node:

```
- While receive Msg(receiver,msg,explanation) do
    Case msg of
    Begin
      abort-operation:          Undo (corresponding S-Tr);
      UNCHECKOUT(O_i):          Release CHECKOUT locks for O_i in the group
                                database;
      start-short-update:       Start a S-Tr to compensate a short update operation;
      D-Tr-backout(D-Tr-id):    Begin
                                inset := Select all objects already checked in by D-Tr;
                                ACTION7 (inset);
                                End; /* msg = D-Tr-backout */
    End. /* case msg of */
```

- Reaction at nodes of affected design transactions:

```
- Decide (either Restore (D-Tr´s state to some external savepoint)
                 or Decide (either Back-Out D-Tr or Continue D-Tr)).
```

Chapter 5

Analyzing Existing Database Recovery Techniques in the Design Environment

After having investigated the requirements on database recovery posed by the various scenarios of the design environment, we can discuss how suitable specific recovery techniques are to support design database systems. Database recovery suitability is usually analyzed on the basis of two criteria: correctness and performance. According to the failure model assumed for design database systems in subsection 4.1.2, a database recovery technique is considered to work correctly in the design environment, if it restores the database to a consistent state in case of processing node failures, stable storage failures, or transaction failures. The basic actions which should be taken by the recovery mechanism in each one of these situations have been already described by means of the recovery protocols presented in chapter 4. In the present chapter, we first discuss the correctness of existing database recovery techniques for the case they are realized in design database systems as the ones considered in this work. This discussion will be based on our failure model and on the set of recovery protocols of chapter 4. In the second part of this chapter, we empirically analyze the performance of existing database recovery techniques for the case they are realized either by the private system at the workstation or by the public system at the server node. This performance analysis is based on the following performance criteria:

- The overhead caused by database recovery activities during normal system operation.
- The cost of recovery in case of failures.
- The volume of recovery information which must be kept in stable storage. By recovery information, we mean the redundant information on the basis of which the recovery algorithm restores the database system to a consistent state in case of failures.

In chapter 6, we present the results of a recovery performance analysis based on simulation which also relied on the criteria listed above and measured recovery costs in terms of achieved system throughput and transaction response time.

5.1 Existing Recovery Techniques and Their Correctness in Design Database Systems

5.1.1 Applying Existing Recovery Techniques to GM1

Nowadays, practically all existing database systems are transaction-oriented, that is, rely on the transaction paradigm to realize reliable database processing environments.

Centralized DBSs implement centralized transaction management while distributed DBSs realize distributed transaction strategies. Moreover, transaction systems can support either conventional transactions or nested transactions. Finally, the nested transaction paradigm can be realized at a single level of the DBS architecture (as it is proposed in [Moss81]) or at various levels of that architecture (as the multi-level transactions in [Wei87b]). Transaction-oriented recovery techniques were first developed to support conventional transactions. Later, they were modified to support extensions to this paradigm.

The GM1 processing model extends the conventional transaction concept to model the design transaction. This extension does not correspond to the nested transaction concept, though. At the private system, the design transaction can be realized as a nested transaction but not all sub-transactions which are started by D-Tr at the server node can be treated as nested (sub)transactions. If, for instance, CHECKIN operations were to be considered part of the nested D-Tr, at commit time their locks would be inherited by D-Tr instead of being released in the public database.

In subsection 5.2.3 where we comment on existing nested transaction-oriented recovery techniques, it will become clear that these techniques cannot distinguish "real" nested sub-transactions (e.g. recovery transactions at the workstation) from sub-transactions which should be treated as conventional transactions at commit time (e.g. CHECKIN operations and short update queries in the public database). Conventional transaction-oriented recovery techniques, on the other hand, cannot support temporary persistency of D-Tr state. Moreover, most of the existing transaction-oriented recovery techniques do not support savepoints, because they are designed to save only data updates (and, maybe, locks). Without modifications, these techniques are cannot capture the state of the computing system at savepoint generation time.

To fully support design transactions in GM1, conventional recovery techniques should be extended to support internal savepoints at the workstation and guarantee temporary persistency of D-Tr state at least at the public system. Logging techniques can be extended through the introduction of new log record types that keep track of the state of the design transaction (e.g. Begin-D-Tr, End-D-Tr). Moreover, short transaction log records must be extended to also refer to the D-Tr on behalf of which the short transaction was started. On the basis of these log extensions, logging and recovery algorithms can be modified to restore the state of D-Tr in case of failures at the server node. Nested transaction-oriented recovery techniques can be modified to exclude sub-transactions of specific types from the nested hierarchie when they commit.

5.1.2 Applying Existing Recovery Techniques to GM2

The GM2 processing model guarantees the atomicity of D-Tr at the public system. Therefore, nested transaction-oriented recovery techniques can cope with this processing model without much modification. In [HHMM88], a nested transaction management strategy was presented which can be applied to GM2. Nested transaction-oriented recovery mechanisms such as the ones proposed in [Moss87] and [ARI89b] support this transaction management strategy without much problems. These mechanisms automatically guarantee temporary persistency of D-Tr state. On the other hand, these mechanisms would have to be extended, though, to cope with internal and external savepoints. Considerations about the realization of savepoint schemes in nested transactions have been presented in [HäRo87].

To be applied to GM2, conventional transaction-oriented recovery techniques should be modified in much the same way they are extended to be applied to GM1. Moreover, these techniques must also be extended to cope with external savepoints. Since external savepoints are generated in the absence of running recovery transactions, the state of the system at savepoint generation time can be easily captured and saved by the recovery mechanism. Recovery mechanisms based on logging can generate external savepoints by simply recording a savepoint record on the log. During external savepoint restoration, all running R-Trs must be backed out and the effect of all R-Trs which committed after the savepoint generation must be rolled back in the private database.

5.1.3 Applying Existing Recovery Techniques to GM3

Since GM3 relies on an object-oriented concurrency control strategy, transaction-oriented recovery techniques cannot support this processing model without far-reaching modifications. Besides the extensions to save and restore the state of D-Tr and G-Tr as well as the extensions to support savepoints, existing recovery mechanisms must be modified to cope with object-oriented synchronization. While nested transaction-oriented recovery techniques support cascading aborts only in a restricted way (i.e. the effects of a sub-transaction are rolled back in the database when its ancestor aborts), conventional transaction-oriented recovery techniques do not cope with cascading aborts at all.

GM3 can be viewed as an extension of the GM1 processing model. While in GM1 D-Tr accesses objects in the public database, it checks objects out of the group database in GM3. The group transaction surrounds the design transaction environment and realizes the group database on the basis of the public database, though. Transaction management (and recovery) can be constructed on the basis of this transaction hierarchy. Object-oriented synchronization must be taken care of only at the group transaction level. At the design transaction level and lower levels, the system can realize the recovery component as in GM1. At the group transaction level, the recovery mechanism can be realized on the basis of the G graph presented in chapter 4.

5.2 An Empirical Performance Evaluation of Recovery in Design Database Systems

As an introduction to the study of suitable recovery techniques for the design environment, we first discuss how transaction management can be realized by the design DBS to control and integrate the various transaction types existent in that environment. Then, we investigate different recovery properties and select those which should improve recovery performance while guaranteeing recovery correctness in the design environment.

5.2.1 Some Transaction Management Alternatives for the Design Environment

On the basis of the reference architecture of chapter 2 and the design processing models proposed in chapter 3, we now discuss how the database system could realize transaction management to support the various transaction types present in the design environment. According to the considerations made in subsection 2.7.3, we will suppose in the following that concurrency control is realized by the DBS on the basis of a locking mechanism.

As already observed in [HHMM88], a single nested transaction mechanism could control the execution of all transactions in the design environment. If we take a server-workstation system based on our reference architecture for design database systems and consider the GM3 processing model, then G-Tr would be the root of a deep nested hierarchy including design transactions, recovery transactions at the workstation, short transactions at the server's object/tuple level, and subtransactions at the server's page/segment level. Nested transaction management in this distributed environment could be accomplished by means of transaction agents (or bookkeepers) as they are described in

[Roth85], [HäRo87], [HHMM88], and [ARI89b]. We shortly explain the idea of transaction agents below.

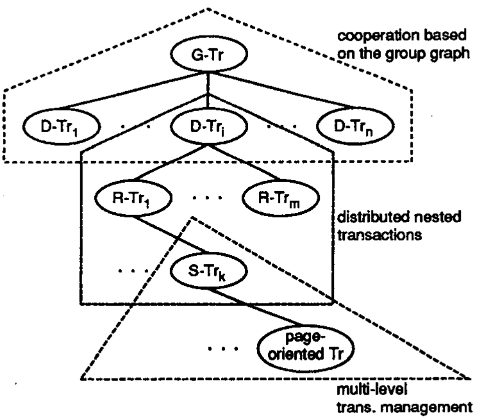
When a (sub)transaction T_1 running in a processing node N_1 starts a subtransaction T_2 at another node N_2, the system automatically creates an agent for T_1 at N_2. It is said that the agent represents T_1 at N_2. Moreover, the system also creates an agent at N_2 for each ancestor of T_1 (if such agents do not yet exist there). If T_2 commits, the locks it holds as well as sufficient recovery information for both redoing and undoing it are inherited to T_1´s agent, before information about T_2 is eliminated at N_2. If N_2 crashes, T_2´s recovery information kept by T_1´s agent is used to redo T_2´s effects in the database. If T_1 aborts at N_1, recovery information kept by T_1´s agent is used by N_2 to undo T_2´s updates in the database. Then, the locks held by T_1´s agent are released, and the agent is discarded. If T_1 commits instead, the locks and recovery information kept by T_1´s agent are inherited to the agent of T_1´s parent transaction at N_2. This step is repeated every time some of T_1´s ancestors commits. When the root transaction commits, its agent is discarded at N_2 and the locks originally held by T_2 are released in the database.

Contrary to the original nested transaction paradigm, the system would have to realize a somewhat modified conversational interface (see [HäRo87]) between G-Tr and its D-Trs to enable G-Tr to see D-Trs´ results as well as inherit D-Trs´locks, yet before design transactions commit. Only by implementing such a mechanism, the system would be able to guarantee design cooperation in a nested transaction environment. As already investigated in [HäRo87], conversational interfaces can strongly increase system complexity.

Representing the whole design effort as a single nested transaction can also reduce concurrency in the system, since page locks acquired by subtransactions running at the server´s page level are held until G-Tr terminates. Even if design transactions can release locks before committing, page locks would be released much later than necessary. Note that this comment applies only to systems where the abstraction level for cooperation between server and workstation is higher than the page (e.g. the record level).

A possibly better solution for controlling concurrent work in a GM3 environment would be to represent the design effort as a set of integrated nested and multi-level transactions. The hierarchy formed by the group transaction and its subordinate design transactions could be realized and managed on the basis of the group transaction graph (G) presented in chapter 4. The whole design transaction at the workstation together with its remote operations at the server´s object/tuple level (e.g. CHECKOUT, CHECKIN) could be realized as a single nested transaction. Finally, the hierarchies formed by transactions at the server´s object/tuple level and those at the server´s page/segment level could be implemented as multi-level transactions so that page locks can be released earlier at the public system.

Figure 5.1 depicts the integrated transaction management environment proposed above. While design cooperation at the group transaction level would be realized in a more flexible way, since the group transaction graph facilitates the control of cooperative work, the nested transaction paradigm would guarantee system reliability in the distributed environment, and multi-level transactions would allow subtransactions to release page locks earlier.



cooperation based on the group graph
G-Tr
D-Tr$_1$
D-Tr$_i$
D-Tr$_n$
R-Tr$_1$
R-Tr$_m$
distributed nested transactions
S-Tr$_k$
page-oriented Tr
multi-level trans. management

Fig. 5.1: Possible transaction management strategy for the design environment

Multi-level transactions as those investigated in [Wei87a] and [Wei87b] constitute a special case of the traditional nested transactions presented in [Moss81]. In the multi-level transaction scheme, each level of the nested hierarchy is associated with a specific layer of the overall system architecture. Therefore, (sub)transactions of different nesting levels process data at different levels of abstraction. Subtransactions process and lock data abstractions which are realized by their related system layers. As a consequence of this architecture orientation, the realization of the multi-level transaction paradigm implies multi-level transaction management, that is, the implementation of one complete transaction manager (consisting of, at least, a concurrency control component and a recovery component) for each of the system layers associated with some level of the multi-level transactions.

DASDBS [Paul87] and MONADS [Frei89] are examples of database systems which realize multi-level transaction schemes. As already discussed in chapter 2, DASDBS realizes a three-layer multi-transaction management. While SMM-transactions lock and process data pages, CRM-transactions work with complex records, and AOM-transactions both lock and process application-specific data objects. Since each transaction level of the nested hierarchy is related to a different sphere of control, locks are not inherited by parent transactions when subtransactions committ. They are simply released to the corresponding transaction manager (in this way, concurrency can be increased in the system).

Another way to guarantee that page locks are released earlier on a server is to realize an open nested transaction mechanism at that node in much the same way it was implemented in System R [Gray81]. Open nested transaction environments differ from multi-level transaction ones in that transaction management is realized at only one abstraction level of the system (e.g. at the tuple level as in System R). At lower levels, the system realizes only some simple synchronization mechanism (e.g. semaphores to control page access).

It should be clear by now that the cooperation between server and workstation could also be realized on the basis of a flat distributed transaction mechanism using some version of the two-phase commit protocol. This strategy can be applied in combination with any of the processing models discussed in chapter 3. In the case cooperation between processing nodes is realized by a flat distributed transaction scheme, each processing node of the system can implement transaction management almost independently from one another. In this case, the concept of a design transaction does not even need to be realized at the workstation. The designer´s work at that node can be seen as consisting of a set of independent recovery transactions running concurrently. Only the public system at the server node would have to be aware of the beginning and ending of the overall design work. This problem could be solved by simply requesting the designer to inform the server every time he either starts or terminates a design work. By applying flat distributed transactions to support cooperation in the design environment, the public system would have to keep track of D-Tr´s state by logging CHECKOUT and CHECKIN operations in stable storage.

If, instead of GM3, the system realizes the GM2 processing model, no group transaction management is necessary. Therefore, the group transaction graph and the algorithms associated with it need not to be implemented. If, on the other hand, the design DBS implements GM1, D-Tr needs not to appear atomic at the public system anymore. As a consequence of this, distributed transaction management can further be simplified in the system.

On the basis of the various alternatives for transaction management in the design environment discussed above and relying on the transaction management model shown in Figure 5.1, we come to the following conclusions:

- Distributed design database systems will probably realize not only one but a set of independent recovery mechanisms which will cooperate to guarantee database system reliability in the design environment.
- Systems realizing GM3 will have to implement recovery mechanisms based on object serializability to support design cooperation and mechanisms based on transaction serializability to support transaction processing at the workstation as well as at the server node. While the latter recover the database state after failures on a transaction basis, the former must be able to restore the database state on an object basis.
- Depending on how transaction management is realized at the workstation and at the server node as well as on how cooperation between processing nodes is controlled, the DBS will have to implement either flat transaction-oriented recovery, or nested transaction-oriented recovery, or even a combination of flat transaction-oriented and nested transaction-oriented recovery mechanisms[1].

5.2.2 Evaluating Conventional Transaction-Oriented Recovery Techniques

In the past ten years, a number of important surveys on database recovery have been published (e.g. [Verh78], [Kohl81], [HäRe83]). These studies can be subdivided into two categories: The surveys which describe existing file recovery techniques without relying on a specific notion of consistency (e.g. [Verh78]) and those works which use the conventional transaction paradigm as the consistency criterion for evaluating recovery techniques (e.g. [HäRe83]).

In this study, we analyze only those existing recovery techniques which have been developed to support transaction systems. This decision has been taken on the basis of two arguments. On one hand, the recovery requirements analysis carried out in the previous chapters has shown that database recovery in the design environment should rely on various transaction types. On the other hand, it is a fact that practically all significant database systems which were developed in the past twenty years (in research centers of the university as well as in the industry) realize transaction-oriented environments.

[1]The term flat transaction distinguishes the conventional transaction paradigm from the nested transaction concept. For a thorough discussion on the differences between flat and nested transactions, the interested reader is referred to [Moss81].

The classification of transaction-oriented recovery techniques presented in [HäRe83] was largely accepted by the database community. It is based on four elementary concepts related to dynamic properties of database systems and specific characteristics of the recovery mechanisms themselves. We briefly explain these four concepts below.

- *Propagation Strategy:* This concept relies on two architectural aspects which are present in most database systems. The first one is concerned with the way data is transferred from main memory to stable storage (i.e. disk). Usually data are written to disk in a page(block)-at-a-time basis. Since either single DML statements or transactions normally affect more than one database page, it is clear that transaction results are not transferred atomically to stable storage. The second architectural aspect to be considered here is that, in fact, the DBS maintains a database hierarchy on disk. At any point in time, the whole set of data stored on disk is considered to form the physical database. Not all these data have a logical meaning to the database system at all times, though. Some of them are, for instance, old versions of updated pages or pages of deleted files. The subset of the physical database that have a logical meaning to the DBS at a specific point in time is called the materialized database. At that time, the rest of the data with some logical meaning to the system are located in main memory. In [HäRe83], the operation of simply writing data from main memory to stable storage (i.e. *write operation*) was distinguished from the operation of integrating updated data into the materialized database (i.e. *propagation operation*).

  If the system writes updated pages back to their original addresses on disk (i.e. update-in-place strategy), transaction results are integrated into the materialized database step-by-step. Therefore, if a system crash occurs before all transaction results have been written to disk, the materialized database becomes inconsistent. That is, only some of the transaction's updates will be reflected in the materialized database. Systems which do update-in-place actually write and propagate updated data at the same time. These systems are said to realize a non-atomic propagation strategy (¬ATOMIC). To prevent the materialized database from becoming inconsistent in the case of a system crash, other systems follow a so-called ATOMIC propagation strategy. In these systems, all transaction updates are first written to a set of addresses in the physical database which are not part of the materialized database. At some later point in time, when all results are stored on disk, already, the DBS integrates these results at once into the materialized database. That is, the system propagates the results atomically. Usually, atomic propagation is achieved by creating a new page table for the materialized database or for parts of it (i.e. database segments). In the new page table, the addresses of old page versions are replaced by the adresses of the corresponding new page versions. The act of changing the page table's address on disk (to reflect the address of the new page table) can be made atomic (see [Lori77]). To distinguish old page versions from their respective new versions, the former are called shadow pages in the literature. Moreover, recovery mechanisms which support ATOMIC propagation are called shadow (page) mechanisms.

  In systems which realize atomic propagation, the state of the materialized database always reflects the results of committed transactions. Since the recovery algorithm relies only on the materialized database and on some recovery information to restore the database state in case of a system crash, global undo recovery is not needed in these systems. This is not true for systems which realize multi-level transactions, though. In those systems, crash recovery must realize global undo operations at higher levels of abstraction (e.g. tuple level) even if propagation is made atomic for lower-level transactions (e.g. page-oriented transactions).

  Atomic propagation can be realized in, at least, two different ways. One alternative is to propagate database updates in a transaction-at-a-time basis. Mechanisms which implement this strategy are called transaction-oriented shadow page mechanisms. On the other hand, the system may propagate results on the basis of disk segments (i.e. sets of contiguous disk addresses). By associating a page table segment to each disk segment on disk, the system may propagate database updates in a segment-at-a-time basis. In this way updates of more than one transaction may be propagated at the same time.

- *Page Replacement Strategy:* This is a much simpler concept which was also used as a basis for the recovery classification in [HäRe83]. If the page-oriented buffer manager may substitute so-called dirty pages in the buffer (i.e. flush pages to disk which contain updates of yet running transactions), the DBS is said to realize a STEAL policy for buffer replacement. On the other hand, if dirty pages may not be replaced in the page-oriented buffer, the system is said to implement a ¬STEAL strategy. Similar to the ATOMIC propagation, the ¬STEAL policy also guarantees that undo recovery operations must be processed only in main memory.
- *EOT Processing:* Systems which guarantee that all transaction updates are stored in the database on disk at transaction commit are said to follow the FORCE strategy of EOT processing. On the other hand, systems which allow transactions to commit, before all their respective updates are stored in the database on disk are said to pursue a ¬FORCE policy. No redo recovery is necessary in database systems which realize the FORCE strategy, because the materialized database on disk always reflects the results of all committed transactions.
- *Checkpoint Strategy:* Checkpointing is the activity of reducing the volume of information needed by the recovery algorithm to restore the database state after a system crash. Checkpoint activity basically relies on propagation activity. Checkpoint mechanisms differ from each other in the time at which the recovery manager must force propagation to take place as well as in how checkpointing affects normal system operation (normal transaction processing activities). In [HäRe83], checkpoint strategies are subdivided into four categories: transaction-oriented (TOC), transaction-consistent (TCC), action-consistent (ACC), and fuzzy checkpoints. TOC reflects the FORCE policy. That is, transaction updates are forced to disk at commit time. TCC implies that checkpointing is not realized in a transaction-at-a-time basis but that all those not yet propagated updates of committed transactions are propagated together at regular time intervals when no user transaction is running in the system. ACC is similar to TCC but it does not require the system to be totally quiescent during checkpoint activity (i.e. user transactions can execute in parallel to checkpoint activity). The system must only prevent update operations from being executed. Fuzzy checkpoint techniques use information about the actual system state to recover it in case of crashes. Since these techniques need to propagate less data than the others, they can allow more parallelism during normal system operation.

Besides the aspects explained above,the survey in [HäRe83] also considered the system level of abstraction where recovery activity takes place as well as the type of recovery information being collected. In principle, the recovery manager may be realized at any abstraction level implemented by the DBS. Recovery information is always collected at the level where the recovery component is located. Thus recovery information can, for instance, be collected at the page level, record level, etc. Moreover, no matter what the level where recovery information is collected, this information can be of one of three types. It can represent either a specific state of some data object (e.g before-images of data pages) or state transitions. Recovery information can represent state transitions either physically (e.g. the set of bits which constitute the difference between the before-image and the after-image of an updated page) or logically (e.g. the DML statement whose execution changed the state of a relational tuple and the index entries associated with it).

Both the level of abstraction and the type of the recovery information determine the cost of the various recovery activities as well as the space on disk needed by the recovery algorithm to a large extent. Let us compare recovery activity at the page level and record level. Assuming ¬ATOMIC propagation with logging, recovery at the page level will usually log more data than recovery at the record level, since the latter writes only updated records to the log file while the former writes the whole pages where these records are located. Depending on how accessed records are stored in the database (e.g.distributed over many pages or concentrated in a few pages), mechanisms which log data at higher levels of abstraction can significantly reduce the log size. Besides that, logging at the page level takes usually longer than logging at the record level. The mechanism at the record level can, for instance, collect before-images and after-images in a log buffer (i.e. a special buffer page) and write them together to the log file in one I/O-operation. The recovery mechanism at the page level must always log complete data pages. Thus, if the transaction updated ten records which are respectively stored in ten different pages and each one of these pages can store ten records, the recovery manager at the page level will request ten I/O-operations to save the transaction´s updates while the mechanism at the record level will, under best conditions, require only one I/O-operation to execute the same task. When recoverying from failures, though, recovery mechanisms at the page level can perform much better than the ones located at higher system levels. In the case of a system crash, for instance, the page-oriented recovery mechanism needs only to read the before-images and after-images located on the log file and reintegrate them into the database (i.e. copy them into the system buffer). The record-oriented mechanism, on the other hand, must read both the updated records located on the log file and the database pages where they were originally stored. While the log file can be read sequentially, database pages must be read with random access.

Based on the architectural aspects described above, the performance of database recovery techniques was analyzed in both [HäRe83] and [Reut84]. While the former work relies on the taxonomy proposed only to informally evaluate the relative costs of transaction-oriented recovery techniques, the latter work uses a set of analytic models to evaluate and compare the performance of various existing recovery mechanisms when they are applied to a centralized database system which realizes a flat transaction-oriented environment. In this experiment, recovery performance was measured on the basis of system throughput (i.e. number of committed transactions per unit of time).

In [AgD85b], the performance of a set of integrated concurrency control and recovery mechanisms was investigated. The authors used the extra transaction costs produced by recovery and synchronization protocols to compare the various mechanisms. The following integrated mechanisms were analytically modeled and compared: log and locking; log and optimistic synchronization; shadow pages and locking; shadow pages and optimistic synchronization; differential files and locking; and differential files and optimistic synchronization.

Results in [Reut84] and [AgD85b] show that recovery mechanisms which are based on shadow pages (as the ones discussed in [Lori77] and [Gray81]) usually perform worse than those mechanisms which realize an update-in-place strategy. This is especially true when the page table of the system is too large to be entirely kept in main memory. The extra I/O-operations necessary to read missing table pages from disk can represent a significant burden during transaction processing. Systems which realize update-in-place strategies need not use the indirection represented by page tables to find page addresses on disk. They directly calculate disk addresses on the basis of the page numbers (i.e. the page number also represents the page´s address on disk).

On the other hand, recovery mechanisms supporting ¬ATOMIC propagation and either STEAL or ¬FORCE must follow the WAL protocol to guarantee database correctness in case of failures. WAL stands for <u>W</u>rite-<u>A</u>head <u>L</u>ogging. This protocol consists of two rules. The first rule guarantees transaction atomicity. It requires that enough information to undo an update operation is saved at some other place on stable storage, before the results of the update operation are propagated to the (materialized) database. The second rule of the WAL protocol guarantees transaction persistency. It requires that enough information to redo the transaction updates is saved on stable storage, before the transaction is committed by the DBS. Therefore, although storage systems supporting update-in-place need not implement page tables, the recovery mechanisms associated with them must maintain redundant information on stable storage (on log files) to recover the system state after failures. Sequential access to log files usually performs better than random access to page table pages, though.

When analyzed in the scope of the design environment, atomic propagation would possibly affect system performance at the workstation and at the server node differently. Since locality of access in design database systems seems to be lower than in business-oriented systems (each small group of designers processes its own data) and the objects processed by design transactions can be much larger than the ones manipulated by business-related transactions, we believe that the public system´s page table can become much larger than it is in conventional DBSs. Therefore, we expect shadow techniques to perform worse than logging techniques at the server node. In case of an integrated information system where both business-related as well as design applications are supported, shadow techniques would probably perform even worse.

At the workstation, on the other hand, the private database stores, at any point in time, data of only one designer. Consequently, recovery transactions belonging to the same D-Tr tend to present a very high locality of access. This high locality together with the fact that the private database is usually much smaller than the public database induce the idea that the private system will not have to deal with very large page tables. Therefore, atomic propagation alone would not reduce system performance at the workstation as it would possibly do on server. Anyway, logging-oriented recovery mechanisms further remain as a good option for the workstation, too. In [HHMM88], for instance, a transaction manager for the PRIMA´s private system is sketched which uses shadow versions of objects to preserve the state they were in by CHECKOUT. Updates to the same object (i.e. atom) are always stored at the same address on disk where the first update to this object was written to. Furthermore, to support savepoints, object updates are also written to a log file. Finally, to use the same object representation in main memory and on disk, the system realizes three indirections (i.e. address tables) to separate logical object addresses from physical ones on disk and in main memory. The mechanism outlined above is an example of hybrid storage system and recovery algorithms which are based on both shadow versions and logging.

Evaluation results in [Reut84] showed that recovery mechanisms which support the STEAL policy perform worse than the ones which support only ¬STEAL in standard DBSs. This can be explained by the fact that the latter mechanisms process undo recovery only in main memory. Consequently, they need to execute less I/O-operations during crash recovery activities. Moreover, these mechanisms save less redundant data (and, consequently, perform better during normal system operation, too), because they need recovery information only to redo transaction results. It should be clear by now, though, that systems which support the ¬STEAL policy require much larger system buffers.

Although main memory prices are continually dropping, it is expected that the size of data objects as well as the number of concurrent transactions in the database system (i.e. the system´s multiprogramming level) will continually increase as the price of computer hardware decreases and database systems evolve to support more complex applications. While the goal of constructing database systems which can process up to 1000 (short) transactions per second has already been achieved, it is expected that design transactions will process data objects of some megabytes. Especially in integrated information systems, it will be difficult to construct main memories large enough to permit the realization of ¬STEAL strategies. Consequently, we believe that recovery at the server node of design database systems will have to support undo operations on disk, too. Maybe, the realization of virtual set/object-oriented buffers could represent a solution to avoid STEAL in design database servers. Although it would still be possible that non-committed updates be written to stable storage during normal system operation, no undo recovery would be necessary in the database on disk . On the other hand, ¬STEAL buffer replacement can probably be implemented at the workstation without problems, since concurrent recovery transactions usually process the same design objects in main memory and the private system normally supports only one D-Tr at a time. As a consequence of the high locality of access and the small number of objects in main memory, this device must not be so large at the workstation as it must be on server.

Another result of the evaluation in [Reut84] is that recovery techniques supporting the ¬FORCE policy perform better than the ones relying on FORCE when commonality[2] is kept high. By high commonality as well as by high locality of access, a subset of the database is accessed much more frequently than the rest of it. In this case, data updated by one transaction have a great chance of being accessed again by the next running transaction. In such an environment, the FORCE policy does not work well because the buffer manager repeatedly flushes data to disk that will be accessed again, soon. In environments where commonality is low, on the other hand, the I/O costs produced by FORCE will probably not represent a heavy burden to the DBS, since the data being propagated at EOT will hardly be accessed again by other transactions.

It is realistic to expect that recovery algorithms supporting FORCE will perform as well as those supporting ¬FORCE at the server node of a design DBS. First, as explained above sets of design transactions usually present lower locality of access. Consequently, a lower commonality can be expected in the page/segment-oriented buffer. Secondly, tests made with the DAMOKLES non-standard database system have demonstrated that data representation mapping operations dominate the costs of short transactions which execute either CHECKOUT or CHECKIN operations at the public system. These (mapping) operations which are processed in main memory take much longer than the extra I/O-operations which can be caused by the realization of the FORCE strategy. Therefore, it can be expected that the FORCE strategy will not represent a significant burden for database servers executing CHECKOUT and CHECKIN operations. Thirdly, if the storage system maintains data belonging to the same object (i.e. the object's subobjects and the description of the relationships among them) clustered in the database and realizes a chained-I/O strategy[3], forcing transaction results to disk at EOT can be acomplished by the system in only a few I/O-operations. This would reduce the burden represented by the FORCE strategy even further.

Since recovery transactions assume the role of recovery points for the design transaction at the workstation [HHMM88], forcing R-Tr results at EOT accelerates crash recovery at that node. Because of the high locality of access at the workstation and the possible absence of mapping activity at the private system, this gain at recovery time would pay only if the operation of propagating updates at EOT could be made cheap. Otherwise, the FORCE strategy will certainly cause system throughput to decrease and transaction response time to increase at the workstation. In R^2D^2, for instance, the recovery manager at the workstation forces updates to disk in parallel with transaction execution [Ries89]. In this way, the system tries to reduce the I/O activity at EOT while realizing FORCE. Maybe, a better alternative in this environment would be to combine parallel logging with normal system operation, since the private system can write updates to the log sequentially while forcing updates to the private database would require random access to disk (and, possibly, some mapping operation, too). Although the strategy in [Ries89] aims at reducing transaction response time, it does not help to increase system throughput.

Results of the evaluation in [Reut84] have also shown that checkpointing is a very important recovery activity in those environments where system crashes are infrequent. Systems not realizing any checkpoint strategy tend to cause very high restart costs, because the number of database updates which have to be repeated after each crash is not bounded in any way. From the various checkpoint strategies analyzed, fuzzy checkpoint

[2]Commonality is defined in [Reut84] as being the probability that a data object will be found in the buffer by the first time the transaction accesses it.

[3]The term chained-I/O stands for I/O-operations which can transfer more than one database page from stable storage to main memory and vice-versa. Usually, the pages which are transferred together in one chained-I/O operation have ascending addresses on disk. For an example of a chained-I/O driver procedure, the interested reader is referred to [WeNP87].

came out as the one which guarantees best recovery performance. Although algorithms which realize fuzzy checkpoint strategies present a higher (software) complexity and usually need to collect and to control more information about the database state than algorithms which implement simpler checkpoint strategies (e.g. the propagation of updated buffer contents), the fact that fuzzy checkpointing can take place in parallel to normal system operation overrides the extra costs and helps to increase overall recovery performance specially in systems with large buffers.

Neither the new system architectures of design DBSs nor the new processing models which are realized by these systems seem to represent new aspects of the database environment that could refute the results of Reuter´s analysis concerning checkpointing. We believe that fuzzy checkpoints will guarantee best recovery performance at the public system at the server as well as at the workstation.

Another result in [Reut84] is that recovery mechanisms which log data at higher levels of abstraction usually perform better than those which log data pages. This was observed in those processing environments where system crashes occur infrequently as well as in those which presented high fault rates. When interpreting these results, one must have in mind, though, that most of the page-oriented recovery mechanisms modeled in [Reut84] either realized no checkpoint at all or implemented either a transaction-oriented or an action-oriented checkpoint strategy. Page-oriented algorithms which do not realize some fuzzy checkpoint strategy may have to force many pages to the database during checkpoint or when recoverying from failures. Moreover, using normal I/O-operations page logging becomes much more expensive than, for instance, blocked record logging. Reuter also modeled and evaluated the DB-Cache technique [ElBa84] which is a page-oriented recovery algorithm that realizes a fuzzy checkpoint strategy and writes pages to the log by means of chained-I/O operations. The DB-Cache model showed one of the best performances of the evaluation, although it logs pages.

We believe that page-oriented logging techniques will not perform worse than other recovery techniques in the design environment. First, the operations to map the main memory representation of highly structured data objects as well as tuple sets onto database pages in design database systems take much longer than the operations to map isolated tuples onto pages in standard DBSs. Consequently, crash recovery at the object-oriented level of a design database system might take much longer than crash recovery at the page level. Second, if data related to the same object are always kept clustered on disk and can be read from as well as written into the database in only a few I/O-operations, the basic drawbacks of page-oriented recovery techniques can be overriden. Besides, page logging is not so expensive (in terms of log space and I/O-operations) if, in most cases, pages respectively contain only data related to the same object. Relying on the observations above, we expect page-oriented recovery mechanisms to perform as well as record-oriented recovery techniques on the server. At the workstation, though, it is quite possible that page-oriented recovery presents a poor performance as it did in Reuter´s evaluation, since the architecture of the private system is similar to reference architecture in [Reut84].

The architecture of the public system and the object-oriented buffer make it possible to realize the server´s recovery manager at the object-oriented level. Although crash recovery at this level of abstraction can take long, the recovery mechanism can reduce the burden during normal system operation by logging sets of object updates as a whole before objects are mapped onto database pages. This would also reduce the size of the log file even further. Besides, the transaction manager at the server could explore the fact that transaction results are saved at the object level to commit transactions even before their updates are mapped onto database pages. A recovery algorithm which realizes this so-called deferred-mapping strategy was proposed in [KARD88]. The main objective of deferred mapping is to reduce transaction response time by committing the transaction before the representation mapping operations related to it are executed. These operations, then, are executed by demon processes in parallel to other transactions. Without a thorough investigation, it is difficult to evaluate the performance of recovery mechanisms which support deferred mapping, though. It is also possible that deferred mapping operations for committed transactions can affect the response time of the next transactions being processed by the public system. In this case, neither response time nor system throughput would be improved.

Although deferred mapping is not of interest for the private system considered in our DBS reference architecture, object-oriented recovery can be considered as an alternative for the workstation, too. Especially recovery algorithms which can either force or log entire (sub)objects using chained-I/O should be investigated in more detail. This algorithms could become even more attractive, if logging activity could take place in parallel to the execution of the respective recovery transaction. The fact that the recovery mechanism logs whole (sub)objects instead of single updated records should not represent a serious drawback, because recovery information would be transferred to the log very fast (through chained-I/O). Moreover, since every log record would contain more than one updated record, the volume of meta information in the log file (e.g. log sequence number, log record type, log record size) would probably decrease.

In [AgD85b] as well as in [Reut84], recovery mechanisms which rely on logging to protect the database against failures came out of the respective performance analysis as the best ones. In [Reut84], these mechanisms have been further specified and investigated in four different processing environments: high update rate with low fault rate, high update rate with high fault rate, low update rate with high fault rate, and low update rate with low fault rate.

From the ten mechanisms analyzed in [Reut84], three came out as the "winners", that is, the ones which guaranteed highest transaction throughput:

- The algorithm described in [Lind79] which logs page entries (e.g. tuples) and can be described on the basis of the classification in [HäRe83] as being (¬ATOMIC, STEAL, ¬FORCE, FUZZY).
- The so-called DB-Cache recovery algorithm proposed in [ElBa84] which logs database pages and can be defined as being (¬ATOMIC, ¬STEAL, ¬FORCE, FUZZY).
- A third recovery mechanism which supports a ¬ATOMIC, STEAL, and ¬FORCE database environment, logs page entries, and implements an action consistent checkpoint strategy (i.e. checkpoints suspend only system operations for record update) at regular time intervals.

The algorithm in [Lind79] differs from the last one above in that it realizes a fuzzy checkpoint strategy. Only updated pages which have not yet been propagated since the last checkpoint are written to disk by the actual checkpoint. DB-Cache logs at the page level and realizes an efficient fuzzy checkpoint strategy which is based on a logically circular log file. Checkpoint takes place only when the circular file becomes full. In this way, checkpoint activity is proportional to user update activity. DB-Cache does not cope well with long-duration transactions, though, because the ¬STEAL strategy forces data updates to be kept in the buffer until transaction commit. If the buffer becomes full with non-committed updates, the system must abort the corresponding transactions. In this way, system throughput can decrease significantly. To solve this problem, it is proposed in [ElBa84] that results of long-duration transactions should be kept in special log files which store enough undo and redo information to support the STEAL strategy. DB-Cache would possibly work well in a public system which realizes GM1 and processes only CHECKOUT and CHECKIN operations and short queries. On the other hand, DB-Cache will certainly perform very poorly if the cooperation between server and workstation is realized on the basis of the nested transaction paradigm. In the GM1 environment, the server processes only isolated short transactions. In a nested transaction environment, short transactions which execute CHECKOUT and CHECKIN operations have to be kept prepared at the server node until the design transaction which started them terminates at the workstation.

5.2.3 Evaluating Nested Transaction-Oriented Recovery Mechanisms

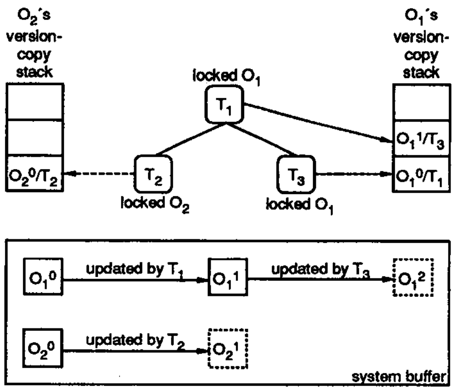
Most recovery mechanisms proposed for nested transactions so far rely on the technique presented in [Moss82]. This technique is an extension of recovery algorithms which use shadow versions of objects to preserve database consistency in flat transaction environments. For every running nested transaction, this technique associates one version stack with each data object accessed by the transaction. Every time a (sub)transaction acquires a write lock on an object, the system makes a copy of the object's actual state and inserts the address of this copy into the respective object stack of the nested transaction. Besides that, the copy's address is also associated with the (sub)transaction accessing the object. After this operation, the (sub)transaction updates the original object version. That is, during normal operation the system follows an update-in-place strategy of propagation. The object's version stack represents only the hierarchy of before-images of the object inside of the nested transaction.

When a (sub)transaction aborts, its associated object versions are used to restore the original states of those objects for which it holds write locks. Depending on its implementation, the recovery mechanism either overwrites the objects' original versions with their respective copies or substitutes addresses of original versions for those of their copies. After that, the recovery algorithm removes those copy addresses associated with the aborting transaction from the respective stacks. When a subtransaction commits, its associated versions are offered to its parent. The parent transaction inherits a version, if it is not yet associated with any other version of the same stack. Versions of objects for which the parent already has associated versions are simply discarded.

On the basis of Figure 5.2, the following example further explains the recovery technique proposed in [Moss82]. While (sub)transactions update database objects in the system buffer, the recovery manager keeps before images for these objects in so-called object-version copy stacks. In the example of Figure 5.2, the recovery mechanism creates a version stack for O_1 when T_1 acquires a write lock on that object. Before T_1 updates O_1 (i.e. creates a new version of O_1), the recovery mechanism inserts a copy of O_1's actual version (i.e. O_1^0) into the nested transaction's copy stack related to this object. After that, T_1 is allowed to update O_1's original copy in the buffer. When T_3 locks O_1 for write, a copy of O_1's new version (i.e. O_1^1) is inserted into the stack and related to that transaction. After that, T_3 is allowed to change O_1 in the buffer. If T_3 aborts, the recovery mechanism simply takes O_1's before-image related to that transaction from the stack and overwrites O_1's copy in the buffer with it. If T_3 commits, on the other hand, the before-image of O_1 related to it in the stack is discarded, since T_1 (i.e. T_3's ancestor transaction) has already updated O_1 before and, consequently, is associated with another before-image of O_1 which is also stored in the stack. If T_1 aborts later on, O_1's before-image related to this transaction is used to restore O_1's original state (i.e. O_1^0) in the buffer. When T_2 acquires a write lock on O_2, the recovery manager notes that there exists no version stack for O_2 related to the nested transaction hierarchy. Thus the recovery mechanism creates a new object-version copy stack in main memory and relates it to both the object O_2 and the nested transaction to which T_2 belongs. Then, a copy of O_2's actual version (i.e. O_2^0) is inserted into that stack and associated with T_2. If T_2 commits, the before-image of O_2 associated with it in O_2's version stack is inherited by T_1. This is done, because although the version of O_2 that was created by T_2 now represents O_2's actual version in the database, O_2's original version (i.e. O_2^0) must have to be restored in the database, if T_1 aborts later on. In the case T_2 aborts, the before-image related to it in O_2's version stack is used to restore O_2's original state in the database. After that, O_2's before-image as well as O_2's version stack can be discarded by the recovery mechanism, since no other transaction of the nested hierarchy holds write locks on that object.

Some implementations of the Moss's version algorithm are described in [Eden82], [LCJS87], and [MuMP83]. Most of these implementations save complete data pages as shadow versions. Also when only the updated parts of data pages are copied as versions (as it is the case by the mechanism presented in [MuMP83]), this recovery technique can become very expensive, since objects can be big and, depending on how deep the nested hierarchy is, many versions of the same object will have to be created by the system for the same nested transaction. Besides, to prevent version management from increasing the volume of I/O-operations in the system, most of the implementations of version-based recovery techniques support only ¬STEAL.

In [ARI89b] where a recovery mechanism for nested transactions based on logging was proposed, the authors used the same arguments presented in [ARI89a] to criticize recovery mechanisms which rely on shadow techniques: very costly checkpoints, extra (volatile and nonvolatile) space overhead for shadow copies, disturbance of the physical clustering of data, and extra I/O-operations due to page faults related to page mapping structures (i.e. page tables). The authors also comment on the (possibly great) system overhead introduced by the algorithms which control object versions inside of transaction hierarchies.



Notes: 1) T_i stands for (sub)transaction i, O_j^v for object j's version v;
2) dotted boxes represent incomplete object versions.

Fig. 5.2:Data structures for the shadow-version recovery example

The recovery mechanism for nested transactions presented in [ARI89b] (i.e. ARIES/NT) relies on the one proposed for flat transactions in [ARI89a] (i.e. ARIES) which, in turn, is based on the recovery technique described in [Lind79]. From the latter, ARIES/NT inherited the log sequence number (LSN) concept which is applied by the system to associate database page versions with log records. From ARIES, ARIES/NT inherited the concept of compensation log records (CLRs) which enables the system to support operation logging (i.e. logical logging of state transitions) and novel lock modes (e.g. lock modes based on commutativity as increment/decrement).

In [Moss87], another logging-oriented recovery technique for nested transactions was presented. This algorithm can work only at the system's page level, though (see [ARI89b] for further explanation). Both ARIES/NT and the recovery mechanism described in [Moss87] are extensions of WAL (write-ahead logging) algorithms for flat transaction environments. They extend those algorithms basically by connecting the log record sequences of transactions pertaining to same nested hierarchies.

In ARIES/NT, for instance, the commit record of every subtransaction is integrated into the backward chain (BW-chain) formed by the log records belonging to the respective parent transaction (see Figure 5.3). Every subtransaction commit record (C-Commit) belongs, therefore, to two different log record chains at the same time: the subtransaction's BW-chain and the parent's BW-chain. C-Commit's field PrevLSN points to the previous log record in the parent's chain while the field LastLSN points to the previous record in the BW-chain of the subtransaction.

By connecting BW-chains of nested transactions, the recovery mechanism builds so-called backward chain trees (BWC-trees) in the log file. Every running nested transaction is represented in the log file by a so-called backward chain forest (BWC-forest) which consists of, at least, one BWC-tree. The first BWC-tree of a forest has the BW-chain of the nested hierarchy's top-level transaction (TL-transaction) as its root. The other possible BWC-trees represent internal portions of the nested transaction which have yet running subtransactions as their roots.

WAL-based algorithms for nested transactions redo as well as undo nested transaction states in case of failures (e.g. system crash) by traversing the transactions' associated BWC-forests in the log file. For each BW-chain visited, these algorithms practically take the same actions as their counterparts in flat transaction environments.

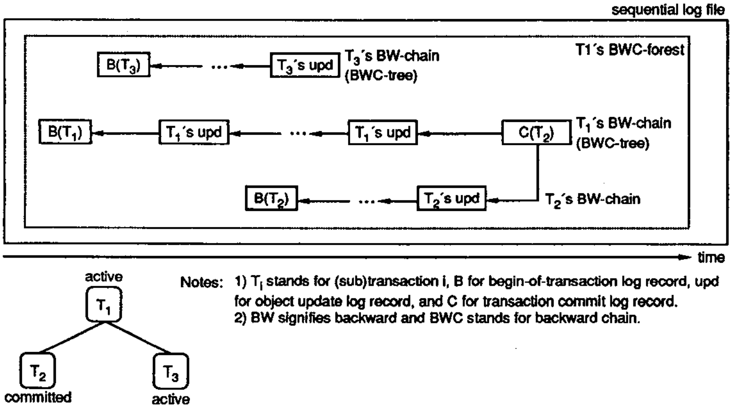

sequential log file
T1's BWC-forest
$B(T_3)$
T_3's upd
T_3's BW-chain (BWC-tree)
$B(T_1)$
T_1's upd
T_1's upd
$C(T_2)$
T_1's BW-chain (BWC-tree)
$B(T_2)$
T_2's upd
T_2's BW-chain
time
active
T_1
T_2
committed
T_3
active
Notes: 1) T_i stands for (sub)transaction i, B for begin-of-transaction log record, upd for object update log record, and C for transaction commit log record.
2) BW signifies backward and BWC stands for backward chain.

Fig. 5.3: Transactions and data structures for the log-oriented recovery example

In [Wei87a], on the other hand, a special recovery algorithm for multi-level transaction environments was presented. This algorithm relies on multi-level logging to cope with transaction backout and system crash. The recovery manager of the lowest system layer logs before-images and after-images of updated data pages. Each higher system layer keeps only undo information related to running transactions which execute at that layer. This information is logged in the form of inverse operations.

To backout a transaction which executes at the system's page level, the recovery mechanism proposed in [Wei87a] restores the state of updated pages by replacing them with their respective before-images which can be found in the system's page-oriented log file. At higher levels, the system rolls back transactions by executing inverse operations in inverse chronological order. The inverse operations related to a transaction can be found in the log file of the system level at which it runs. Inverse operations are executed as part of the transaction being aborted. Since inverse operations at one system layer are partially realized by operations (i.e. transactions) at lower system levels, locks at those levels have to be acquired again. As a consequence, transaction backout in higher layers may involve lower-level transactions in block as well as deadlock situations. Moreover, to resolve deadlock situations, the system may have to backout inverse operations (possibly indefinitely), too.

After a system crash, the multi-level recovery mechanism recovers the database state by serially recoverying the state of each system layer (from the lowest to the highest layer). First, it brings the database to a lowest-level transaction consistent state by undoing the effects of either running or already aborted page-oriented transactions and redoing the updates of committed page-oriented transactions. Then, in each higher system layer transactions which were running by the time of the crash are completely backed out (through the execution of inverse operations).

Besides presenting the drawbacks related to the execution of inverse operations at higher levels (which actually represent side-effects of the associated multi-level concurrency control strategy), the recovery algorithm proposed in [Wei87a] shows a poor performance due to logging redundant recovery information at several system levels. The control and synchronization of the different log files surely represent an extra burden to transaction processing activities (see [Wei87b] for a thorough discussion on these topics).

If the design DBS realizes the designer's work (i.e. the design transaction) as a distributed nested transaction, recovery at the workstation as well as on server should rely on logging. Based on the recovery analysis presented above, we can conclude that nested transaction-oriented recovery techniques which rely on shadow copies would require very large main memories and stable storage space at both server and workstation, since design objects can be very large. Besides, the processing overhead necessary to manage internal version hierarchies in nested transactions would certainly represent a significant burden during normal system operation. Especially at the workstation where both processing capacity and storage space are restricted, internal version management would possibly reduce system throughput as well as increase transaction response times.

As already discussed in section 5.1, the public system can improve concurrency on server either by realizing an open nested transaction mechanism or by implementing a multi-level transaction scheme at that node. Because of the problems associated with multi-level concurrency control as well as multi-level recovery, we believe that the open nested transaction alternative should be preferred. In this environment, the recovery mechanism can be realized at a single level of abstraction (e.g. tuple level). This can significantly simplify both the construction and operation of the recovery mechanism on server.

Object-oriented recovery mechanisms, too, can rely either on logging (i.e. support update-in-place) or on shadow versions (i.e. support atomic propagation). Object-oriented techniques based on shadow versions represent a straightforward solution to object-oriented recovery at the server node as well as at the workstation. On the other hand, these techniques cause the drawbacks discussed above for transaction-oriented recovery. If, for instance, design transactions are realized as nested transactions at the workstation, the shadow version algorithm described in [Moss82] can be modified as to guarantee object-oriented recovery at that node. If D-Tr is requested by G-Tr to roll back modifications done on a specific object, the private system must proceed as follows. First, all running recovery transactions which have locked the object or are waiting to lock it must be aborted. Then, the object's actual version must be deleted in the private database. After that, the whole shadow version stack associated to the object must be destroyed and its storage space released in the private database. Finally, D-Tr must UNCHECKOUT the object in the public database.

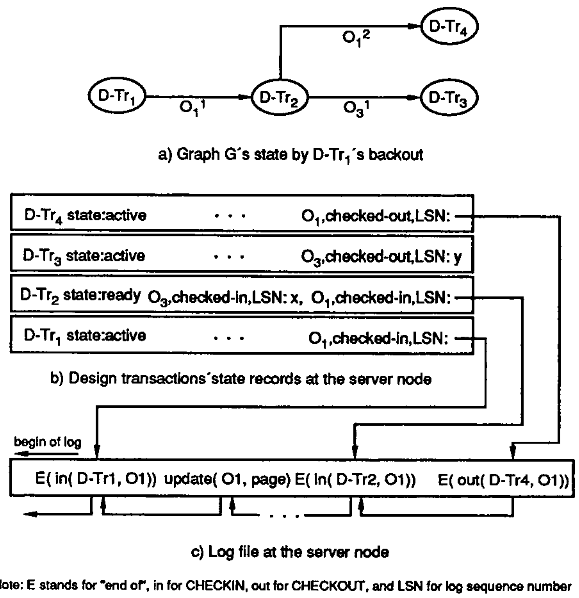

D-Tr4
O1 2
D-Tr1
O1 1
D-Tr2
O3 1
D-Tr3
a) Graph G's state by D-Tr1's backout
D-Tr4 state:active ... O1,checked-out,LSN:
D-Tr3 state:active ... O3,checked-out,LSN: y
D-Tr2 state:ready O3,checked-in,LSN: x, O1,checked-in,LSN:
D-Tr1 state:active ... O1,checked-in,LSN:
b) Design transactions'state records at the server node
begin of log
E(in(D-Tr1, O1)) update(O1, page) E(in(D-Tr2, O1)) E(out(D-Tr4, O1))
c) Log file at the server node
Note: E stands for "end of", in for CHECKIN, out for CHECKOUT, and LSN for log sequence number

Fig. 5.4: Example of recovery information associated with G-Tr at the server node

To avoid the drawbacks of shadow version techniques, the DBS can realize object-oriented recovery on the basis of logging techniques. In the following, we propose an extension of ARIES/NT which also supports object-oriented recovery. We explain this extension by supposing that the group transaction environment is realized by the public system on the basis of a nested transaction mechanism implemented at the server's record level. Moreover, we assume that design cooperation is controlled on the basis of the G graph presented in chapter 4. Besides maintaining one backward chain on the log for every transaction of the nested hierarchy formed by G-Tr as well as by its subordinate D-Trs and their respective short transactions, the recovery mechanism also keeps special backward chains on the log which respectively associate log records for the same object. Object-oriented log record chains are not bound to transaction-oriented chains. That is, log record chains for updated objects can span log chains for transactions. For every running D-Tr (i.e. a D-Tr which has neither committed nor aborted), the transaction manager keeps a list of all objects that have been checked out by this transaction in its D-Tr state record at the server node. Every object entry in this list contains, at least, the actual status of the object (i.e. checked-out, unchecked-out, checked-in) and the address of the youngest log record for this object which has been written on D-Tr's behalf.

We explain D-Tr backout in a design cooperation environment using the scenario shown in Figure 5.4. Figure 5.4 (a) depicts the state of the graph G at the time D-Tr$_1$ decides to abort. Both D-Tr$_2$ and D-Tr$_4$ have seen non-committed results (i.e. O$_1$'s version 1) of the aborting transaction. While D-Tr$_2$ is already in the ready state, D-Tr$_4$ is running yet. By analyzing G, the recovery mechanism concludes that D-Tr$_1$'s effects can be removed from the group database, if the original version of O$_1$ is restored therein. To achieve this goal, the recovery algorithm takes the following steps. First, it identifies the transactions which must be involved in this operation. Then, it analyzes the state records of those transactions to decide which operations each one of them must execute. D-Tr$_4$ must be requested to check O_1 back into the group database without changes. D-Tr_2 must be brought back into the active state and its owner must be informed that $O_1{}^1$ has been invalidated. Finally the state O_1 was in before D-Tr_1 checked it back into the group database must be restored. This last operation is executed by the recovery algorithm itself. It first identifies the last log record associated with O_1. It finds the address of this record in D-Tr_4's state record. By following the backward chain for O_1 on the log, the recovery algorithm keeps on restoring O_1's older states until the Begin-of-Checkout record related to both D-Tr_1 and O_1 is found on the log file. As with ARIES/NT, corresponding compensation log records are written to the log as O_1's backward chain is being processed.

Nested transaction-oriented recovery techniques applied to the workstation must also be extended to cope with design cooperation. We can explain this on the basis of the example given above. Suppose D-Tr_4 had already modified O_1 at the workstation by means of a set of successful recovery transactions as it was requested to uncheckout that object. If the workstation crashes after the uncheckout operation is completed, the recovery mechanism will try to redo the effects of those committed recovery transactions which updated O_1 in the private database. This mechanism should not do that, though. First, O_1 cannot be found in the private database anymore. Secondly, the effects of those recovery transactions which only updated O_1 should not be recovered. This problem can be solved by writing a special uncheckout record for O_1 to the log file at the workstation. By reading the log file backwards during crash recovery, the recovery manager finds this record before it reads the respective commit records for those recovery transactions which processed O_1. Therefore, it can avoid redoing the effects of those transactions when it finds their records on the log.

5.2.4 Further Recovery Classification for the Design Environment

Besides the architectural properties considered in [HäRe83], recovery in the design environment can be analyzed under various other aspects. Relying on our study of recovery requirements, we can identify, at least, three other important classification criteria for recovery in design database systems. In the following, we both present and discuss these criteria on the basis of which recovery can further be analyzed.

- *Node Cooperation:* Reflecting the kernel architecture of design database systems as well as the server-workstation computer configuration of the design environment, the DBS can either realize a systemwide integrated recovery mechanism or implement isolated recovery managers in each of the system's processing nodes. Integrated recovery algorithms as the one proposed in [KaWe84] explore the failure independence property of processing nodes to improve overall system reliability. For instance, to cope with both small disks and disk crashes at the workstation, the private system's recovery manager can send transaction updates to be saved at the server node from time to time. On the other hand, in case of disk crashes on server, the public system can ask the private systems at workstations to send the CHECKOUT versions of those objects actually being processed there.

  While integrated recovery mechanisms can improve reliability in the design environment especially in the case of hard failures, they would possibly represent an extra burden for the communications subsystem and reduce the server's performance, since the public system would have to support recovery at the workstations besides executing its own tasks. The extra burden on server would grow with the number of workstations in the system. Integrated recovery could alternatively be realized only among workstations, though, to prevent the extra burden on server. The private system at the workstation could, for instance, use the designer's think times to broadcast transaction updates to other workstations. In case of a disk crash, the private system would request the data to be sent back. Since possibly more than one other workstation stored the updates (sent by the broadcast operation), the probability that all of them be off at the time of the crash is not high.

Isolated recovery managers can be designed to better cope with special characteristics of specific processing nodes. In systems where workstations have enough disk space to store the whole private database (with all its versions), the alternative of isolated recovery managers for server and workstation would possibly improve overall system performance, since less communication between server and workstations would be necessary. On the other hand, isolated recovery algorithms, too, can cope with disk crashes at the workstation. For instance, from time to time the recovery manager at the workstation could write transaction updates to normal data files on server by means of remote update queries to the public database. The log files in the public database could be created by the designer and associated to his D-Tr at the beginning of the design work.

- *Transaction Cooperation:* This classification criterion distinguishes recovery techniques which are based on transaction serializability from those which permit transaction cooperation (e.g. algorithms supporting the predicatewise two-phase protocol). Database systems supporting design cooperation as it is defined in GM3 must implement a hybrid recovery mechanism which can guarantee transaction serializability for short and group transactions in the public database as well as for recovery transactions in the private database, while supporting object serializability for design transactions in the group database. Recovery subsystems which support either GM2 or GM1 can exclusively rely on transaction serializability.

  If the DBS realizes GM3, the workstation´s recovery mechanism must be able to roll back work on an object basis, too. While a transaction-oriented recovery mechanism must be applied at the recovery transaction level at the workstation, an object-oriented recovery mechanism must support design transaction management at that processing node. In the following subsection, we propose an alternative to integrate transaction-oriented recovery with object-oriented recovery at the workstation. Similar to transaction-oriented recovery mechanisms, object-oriented recovery techniques can also be classified on the basis of the architectural concepts considered in [HäRe83].

  Group transaction management will probably be realized by the public system at the server node. In this environment, recovery mechanisms which realize deferred mapping will probably perform well. Since design transactions exchange non-committed object versions by first checking them into the group database, recovery based on deferred mapping could help saving mapping operations at the server node in, at least, two distinct ways. We explain this on the basis of a recovery mechanism which logs object versions at the object-oriented level of the public system. The designer executing the CHECKIN operation could in addition inform the public system that the version being checked in will soon be checked out by another designer (we assume that designers belonging to the same group are always acquainted with each other´s work and intensions). In this case, the system would save the updated object in its actual representation (i.e. level of abstraction) and commit the CHECKIN operation. When the other designer starts his (expected) CHECKOUT operation, the server would simply read the object from the log file and send it to the workstation without changing its representation. In this way, the public system would avoid two actually unnecessary (possibly long-duration) mapping operations. The public system can also reduce the number of mapping operations at the server node by automatically waiting to map object versions which have already been checked in until the design transactions that created them commit. If any of these objects is checked out and checked back in by another design transaction before the transaction which created it commits, only the version checked in later needs to be mapped onto database pages.

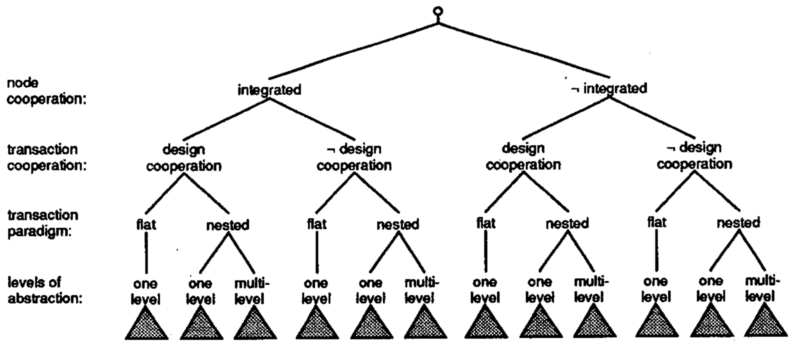
- *Transaction Paradigm:* At various levels of the design environment, transaction management can be realized in either one of two ways: on the basis of the (conventional) transaction paradigm (and its distributed transaction extension) or by following the nested transaction concept. The whole transaction hierarchy forming the design transaction, for instance, can be either implemented as a distributed nested transaction or partitioned in such a way that it forms a nested transaction at the workstation which can start a set of flat transactions on server. While nested transaction-oriented recovery techniques can cope with both flat and nested transaction environments, flat transaction-oriented recovery mechanisms (as the ones investigated in [HäRe83], [Reut84], [AgD85a], and [AgD85b]) can only support flat transaction systems.

Nested transaction-oriented recovery mechanisms can further be classified on the basis of two orthogonal criteria: recovery strategy and level of abstraction. Concerning the recovery strategy, these mechanisms rely either on shadow versions of data or on logging. Shadow version algorithms provide each transaction of the nesting hierarchy with the information necessary to recover its state in case of failures independently of other transactions in the hierarchy. Logging mechanisms maintain recovery information integrated and centralized.

Recovery algorithms which support (conventional) nested transaction environments usually operate at only one level of abstraction (e.g. page level). If the system realizes a multi-level transaction scheme, though, the recovery manager must operate in all those system levels where transaction management takes place. In the next subsection, we analyze various existent nested transaction-oriented recovery techniques in more detail.

5.2.5 Summarizing the Conclusions of the Empirical Recovery Evaluation

In the present section, we have first analyzed recovery for design database systems on the basis of flat transaction-oriented recovery mechanisms. For this part of the study, we based our considerations about recovery performance on the classification of recovery techniques proposed in [HäRe83] as well as on the evaluation results of recovery performance studies presented in [Reut84] and [AgD85b]. Then, we have extended the classification of recovery mechanisms for the design environment by introducing three other classification criteria, namely how recovery mechanisms located in different processing nodes cooperate, if the algorithms support transaction cooperation, and the transaction paradigms which are followed by different recovery algorithms. Figure 5.5 shows the hierarchy formed by the proposed classification criteria. The triangles in this figure represent instances of the hierarchy of concepts proposed in [HäRe83] which is illustrated in Figure 5.6. Recovery mechanisms realized at the system's object-oriented level can further be subdivided into two categories: the mechanisms which support deferred update and the ones which do not.



Notes: 1) The triangles represent the recovery classification proposed in [HäRe83] which is depicted in Figure 5.6.
2) The last three classification criteria, namely transaction paradigm, levels of abstraction, and the classification in [HäRe83] refer to each recovery subsystem being implemented in the design environment.

Fig. 5.5: Classification of recovery mechanisms in the design environment

In the following, we summarize the main results of the empirical recovery evaluation carried out in the previous subsections:

- Concerning the possible cooperation of recovery algorithms in different processing nodes, we believe that isolated recovery mechanisms on server and at the workstation would perform better than integrated recovery mechanisms. The latter may cause a significant extra burden for both the communications subsystem and the public system on server. On the other hand, integrated recovery mechanisms involving only workstations may be able to improve reliability at the workstation without reducing system performance very much.
- If the DBS supports design cooperation, the recovery mechanism at the server must combine object-oriented with transaction-oriented recovery actions to cope with the hybrid recovery environment represented by GM3. Otherwise, only transaction-oriented recovery techniques must be realized by the public system. Depending on the transaction management strategy selected, either a single-level or a multi-level recovery mechanism will have to be realized by the public system.
- The recovery manager at the workstation must support transaction serializability only.
- Recovery techniques which support deferred mapping should perform well in the GM3 environment. In the other processing environments (i.e. GM1 and GM2), it is not clear if the benefits of deferred mapping will override its associated extra cost.
- Nested transaction-oriented recovery should be realized in the design environment on the basis of logging. Nested recovery techniques following the WAL principle are expected to perform much better than those which rely on hierarchies of shadow versions.
- Multi-level recovery mechanisms will probably perform worse than single-level algorithms on server.
- We believe that both page and tuple logging will perform similarly on server, since processing time will be dominated by mapping operations at that node. On the other hand, recovery mechanisms which support object-oriented logging and deferred mapping on server should be investigated in more detail.
- Techniques which collect recovery information at the object level can possibly work well at the workstation, too, especially in the presence of expensive mapping operations (as it is the case in R^2D^2 [KeWa88]).
- Tuple logging will surely perform better than page logging at the workstation when no expensive mapping operations take place at the private system.
- Recovery techniques which support atomic propagation will probably perform better at the workstation than at the server node. Non-atomic propagation seems to be the best choice for the public system. On the other hand, it is not clear if atomic propagation performs worse than non-atomic propagation at the workstation when the private system realizes a flat transaction environment.
- It seems more realistic to think that the page-oriented buffer manager will realize the STEAL strategy at the server node. In this case, recovery at that node will also have to cope with undo operations on disk. Some new DBS prototypes supporting buffer hierarchies avoid STEAL by realizing high-level virtual system buffers, though (see [KARD88]). Although these systems cannot prevent non-committed data from being flushed to stable storage, these data are not propagated to the materialized database. Consequently, undo recovery takes place only in main memory.
- ¬STEAL can be realized by the private system at the workstation without much problems, since concurrent recovery transactions present high locality of access and the system processes only one D-Tr at a time. If ¬STEAL is implemented at the workstation, recovery activity could become much cheaper at that node.
- We believe that recovery mechanisms relying either on FORCE or ¬FORCE will perform alike on server due to the expensive mapping operations which are executed at that node. On the other hand, recovery based on ¬FORCE will probably perform

better at the workstation, especially in combination with parallel logging or parallel flushing strategies.

- We expect fuzzy checkpoint techniques to reduce recovery overhead on server as well as at the workstation much in the same way they do it in business-oriented database systems.

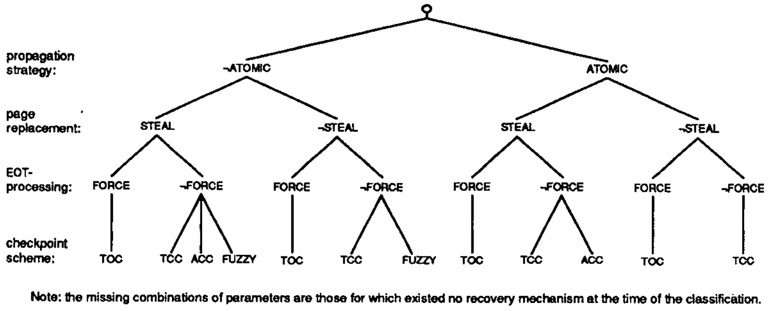


Fig. 5.6: Classification of transaction-oriented recovery schemes proposed in [HäRe83]

Chapter 6

Simulating Recovery Techniques in a Design Database System

6.1 Establishing the Goals of the Performance Evaluation

In the previous chapter, we have analyzed database recovery in the design environment on the basis of a set of database system concepts. Some of these concepts were already used on the classification and evaluation of recovery techniques for centralized business-oriented database systems. Others were selected because they represent specific architectural properties of design database systems. Relying on the results of the analysis carried out in the last chapter, we come to the following conclusions:

- Transaction-oriented recovery techniques as they are known from business-oriented environments cannot cope with all types of design processing environments. These techniques must be modified to support design transaction cooperation, since cooperating transactions also require object-oriented recovery.
- Compared with their performance in business-oriented database systems, transaction-oriented recovery mechanisms may perform differently in design database systems. This is a consequence of the different architecture and processing models realized by design database systems.
- The same recovery technique may perform differently depending on the DBS´s processing node at which it is realized. This is a consequence of the different architectures and processing models realized by different processing nodes (e.g. server and workstation) of the design environment.
- New architectural properties of design database systems (e.g. more levels of abstraction than conventional DBS architectures, hierarchies of system buffers) make the realization of new database recovery mechanisms in the design environment possible (e.g. recovery at the object-oriented system level, recovery supporting deferred mapping).

Probably, the architectural aspects which most affect recovery performance in design database systems are the expected long-duration mapping operations at the server node and the database hierarchy distributed over the server and the workstation. In the last chapter, we empirically evaluated the performance of different recovery mechanisms at the workstation and at the server node on the basis of these architectural aspects. From that evaluation, it became clear that some evaluation results of former recovery performance analysis based on business-oriented database systems will probably not be observed in the design environment. On the other hand, we believe that new recovery techniques for design database systems should be investigated in more detail.

As noted in chapter 5, there already exists a number of independent studies concerning the performance of recovery techniques for database systems. Most of these studies have investigated recovery performance only for centralized database system architectures supporting conventional, business-oriented transaction environments, though (e.g. [Reut84], [AgD85b]). In [AgD85a], recovery mechanisms for multiprocessor database machines were investigated. Although this investigation relies on a distributed database system architecture, recovery has been analyzed in a conventional transaction environment. The evaluation reported in [Wei87b], on the other hand, possibly is the only one where database recovery performance has been investigated in a multi-level transaction environment. Nevertheless, this study considered only conventional transactions (i.e. long-duration transactions were not simulated), and modeled the multi-level environment without a buffer hierarchy. The work in [Wei87b] mainly compared multi-level transaction management (as a whole) with single-level transaction management for centralized database systems.

In this chapter, we present a simulation analysis of recovery performance in design database systems. On the basis of this analysis, we tried to quantify the results of the empirical evaluation carried out in the previous chapter. We concentrated our efforts on the investigation of both recovery techniques which were expected to perform differently when applied to the design environment (e.g. recovery based on FORCE) and those recovery mechanisms which can be realized at the object-oriented level of the design database system. Moreover, we evaluated recovery performance on the basis of a database system which supports business-oriented as well as design transactions, since we believe that non-standard database systems will usually have to cope with integrated information systems, that is, systems where totally different applications which can process common data might have to be supported at the same time.

We decided to simulate only the public system on server, because the new architectural characteristics of design database systems in which we were mostly interested are realized by that subsystem. The architecture of the private system, on the other hand, is either an extension of that of the public system (i.e. when the storage system supporting the application-oriented layer at the workstation is either similar or even a copy of the DBS kernel software[4]) or it is similar to the architecture of a centralized business-oriented DBS supporting conventional transactions. In the first case, recovery requirements at the workstation can be derived from those posed by the public system at the server node. In the second case, recovery at the workstation can be compared with recovery in centralized database systems and, for the latter, there already exists a number of performance studies. Furthermore, we believe that the performance results achieved by the server simulation will help us to analyze recovery performance at the workstation in a future work.

By analyzing the performance of recovery mechanisms at the server node of a design database system, we were mainly interested in investigating the following questions:

- How well recovery mechanisms can cope with environments where totally different transaction classes are supported.
- To which extent recovery can benefit from the buffer hierarchy realized by the system.
- In which level of abstraction recovery mechanisms should be realized in the system to guarantee better performance.
- How recovery mechanisms can help to improve overall system performance in the design environment.

Most of the simulation study were based on the GM1 design processing model. GM3 was simulated only to test the performance of deferred mapping in environments supporting design cooperation. Since GM2 constitutes a special case of GM1 at the server node, we

[4]An example of such an architecture can be found in [DeOb87].

believe that most of the results obtained through the simulation study can be extended to server nodes which realize GM2.

Database recovery in the evironment described above was investigated on the basis of three performance criteria: the way different algorithms affect system throughput[5]; how transaction response time is affected by recovery activity; and how much stable storage space is needed by each mechanism to save recovery information during normal system operation. Note that none of the already published recovery performance studies has investigated either the effect recovery mechanisms have on transaction response time or how much data they need to keep in nonvolatile storage. Although the burden represented by recovery activity in the overall transaction cost was investigated in [AgD85b], response time during system operation has not been directly modeled[6]. On the other hand, we believe that recovery mechanisms supporting the design environment will need much more space on stable storage than recovery mechanisms in conventional DBSs do. Therefore, especially for small and medium size design systems recovery algorithms which save less data (in terms of number of bytes kept on stable storage) will be of greatest interest (see [Härd87] for more explanations on this subject).

To better analyze how different recovery activities affect system performance, database recovery was investigated in four different environments. These environments were selected on the basis of the various failure modes considered by the failure model derived in chapter 4. The simulation of four different system environments simplified the evaluation of both the recovery cost associated with each individual failure mode and the burden represented by recovery activities during normal system operation. We present the four environments below.

- *Normal System Operation (NR):* In this environment, transactions always execute normally. That is, transactions never need to be aborted. Moreover, the system never fails (i.e. no crashes occur). Finally, when analyzed in this environment, recovery mechanisms generate no checkpoints. By the simulation of this environment, we could evaluate the burden represented by recovery activities during normal system operation as well as the volume of recovery information kept on stable storage when no checkpoint activity takes place.
- *Abort Environment (AB):* As the name suggests, in this environment transactions can be both rolled back and restarted until they execute to completion. On the average, some 10% of all transactions executed in this environment abort, at least, once. As with the normal environment, neither crashes occur nor checkpoints are generated in the abort environment. By the simulation of this environment, we were mainly interested in evaluating recovery cost for transaction backout.
- *Checkpoint Environment (CH):* In this environment transactions always execute to completion, no crashes occur, and the recovery mechanisms are allowed to generate checkpoints. By the simulation of this environment, we could evaluate the cost associated with checkpoint activity during normal system operation. Moreover, we could also analyze how checkpoint activity contributes to reduce the volume of recovery information kept on stable storage.
- *Crash Environment (CR):* This environment produces system crashes regularly (i.e. typically, 4 to 5 crashes per execution of 2000 transactions). Transactions never abort and no checkpoints are generated in this environment. By the simulation of this environment, we were mainly interested in the evaluation of the cost associated with crash recovery activity.

[5]System throughput was measured on the basis of the number of committed transactions per unit of time.

[6]In [AgD85b], dynamical properties of the operating system supporting the DBS as, for instance, multiprogramming level and how the execution of one transaction affects the execution of others were not taken into consideration by the analytical models.

Besides defining different system environments, we generated a set of different transaction loads by changing the values of some important load parameters. Different load types basically differ from one another in the relation between the number of business-oriented and design transactions they contain as well as in the update transaction rate they present. Later in this chapter, we present the various simulated transaction loads in more details (see section 6.3). Through the integration of different load types with different environments, it was possible to evaluate recovery performance in a set of processing scenarios which possibly represent most of the real scenarios in a design environment realizing the GM1 processing model.

The selection of recovery mechanisms to be simulated followed the considerations about recovery techniques made in chapter 5 and reflected the main goals of the performance analysis. We evaluated only non-integrated recovery algorithms. These algorithms guarantee correctness only for databases at the server node (e.g. the public and the group database). They are not related to recovery activities at the workstation. In most of the simulation experiments, the selected recovery algorithms supported non-cooperative design environments (e.g. GM1). In some experiments, we simulated design cooperation to some extent, though. Since we simulated the GM1 processing model most of the time and did not model the designer work at the workstation in much detail, we selected only flat transaction-oriented recovery techniques for simulation. Although the construction of a simulation model for some of the selected techniques has proven to be a complex task, the development of models for nested transaction-oriented recovery techniques would certainly be much more complex.

We were most interested in comparing the performance of recovery mechanisms which are implemented at different system levels. Therefore, we selected two mechanisms which work at the page level, one which logs tuples and data records, and one which works at the object-oriented level of the public system and supports deferred mapping. According to the considerations in chapter 5, all recovery algorithms selected support non-atomic propagation. To investigate the behavior of FORCE in design database systems, we selected one technique which supports this EOT strategy. The other three recovery algorithms support ¬FORCE. All recovery algorithms selected for simulation realize some kind of checkpoint. Finally all selected techniques support the STEAL policy of buffer replacement.

We analyzed the algorithm based on deferred mapping in two different operating system environments. In the first one (OSE1), the system's multiprogramming level is kept constant concerning deferred mapping operations. That is, the system waits until the mapping activity for a committed transaction is completed, before it takes another transaction from the ready queue and starts executing it. In the second operating system environment considered (OSE2), new transactions are taken from the ready queue and processed by the system as soon as old transactions are committed (i.e. even before mapping operations related to already committed transactions terminate).

Relying on the extended taxonomy of chapter 5, the algorithms chosen for analysis can be characterized as follows.

- *REC1:* (¬INTEGRATED, ¬COOPERATION, FLAT, PAGE-LEVEL, ¬ATOMIC, STEAL, ¬FORCE, ACC). This algorithm logs before and after images of updated database pages. It is based on the undo/redo recovery mechanism described in [BeHG87]. It generates checkpoints either at regular time intervals or when the log achieves a previously specified length. During checkpoint, updated pages which have not yet been propagated since the last checkpoint are written to the database on disk.
- *REC2:* (¬INTEGRATED, ¬COOPERATION, FLAT, OBJECT-LEVEL, ¬ATOMIC, STEAL, ¬FORCE, FUZZY). This algorithm saves updated versions of design objects and tuple sets before these abstractions are mapped onto database pages by the public system. It implements the log as a ring file and asks the buffer manager to flush pages (in the buffer) to disk only when the ring file is full (as with the DB-Cache algorithm in

[ElBa84]). REC2 is based on the deferred mapping version of the recovery mechanism described in [KARD88].

- *REC3:* (¬INTEGRATED, ¬COOPERATION, FLAT, TUPLE/RECORD-LEVEL, ¬ATOMIC, STEAL, ¬ FORCE, ACC). REC3 is a modified version of the recovery algorithm proposed in [Lind79]. It logs before and after images of updated data records (i.e. atomic objects) and tuples, after updated objects and tuple sets are mapped onto database pages. Opposed to the algorithm in [Lind79], REC3 realizes an action consistent checkpoint strategy[7].
- *REC4:* (¬INTEGRATED, ¬COOPERATION, FLAT, PAGE-LEVEL, ¬ATOMIC, STEAL, FORCE, TOC). This algorithm works at the page-level, supports FORCE and realizes a transaction-oriented checkpoint strategy. That is, transaction updates are always integrated into the database on disk at transaction commit.
- *REC5:* (¬INTEGRATED, ¬COOPERATION, FLAT, OBJECT-LEVEL, ¬ATOMIC, STEAL, ¬FORCE, FUZZY). Actually, REC5 and REC2 represent the same recovery algorithm. They differ from each other in that REC2 is evaluated in the OSE1 operating system environment while REC5 is analyzed in OSE2.

Through the simulation of REC2 and REC5 we could evaluate the performance of recovery algorithms which execute at the object-oriented level of the server and support deferred mapping. On the basis of these recovery algorithms, we were able to analyze the performance of deferred mapping in the GM3 environment, too. Through the simulation of REC1, we could evaluate the performance of page-oriented logging mechanisms in a processing environment where representation mapping operations dominate transaction processing time and the operating system supports chained-I/O. Finally, we simulated REC4 to analyze the performance of recovery mechanisms which support FORCE in the design environment.

6.2 A Simulation Model for Recovery Performance Evaluation

In this subsection, we comment on the simulation model on the basis of which the performance of the recovery mechanisms presented above have been analyzed. A complete description of this model including implementation details can be found in [Ioc89a] and [Schm89].

As we decided to model recovery activity in the design environment, the question of what methodology to follow arose. As already noted in [Wei87b], to be able to mathematically handle analytical models, designers are often forced to make unrealistic assumptions about system characteristics. Furthermore, systems which present a relative large number of independent variables cannot be evaluated by means of analytical models at all. Pure simulation models, on the other hand, usually are simpler to develop and realize, but they can also lead to an unrealistic description of the system, in case designers choose incorrect values for model parameters.

The best method for analyzing the performance of computer system components seems to be the evaluation of their functionality in real systems on the basis of real transaction loads. In some studies where new system architectures are to be analyzed in an existing transaction environment, simulation models were combined with so-called real transaction reference strings (e.g. [HäPR85]). These are logs containing sequences of write and read operations executed by real transactions running on a real computer system. The reference strings, then, are used as input in computer simulation models. As we developed our system model, there was neither an existing design DBS similar to the one we wanted to

[7] This algorithm was identified as '2.2' in [Reut84]

investigate nor a reference string of design transactions. Therefore, we decided to develop a simulation model for our design environment and generate transaction loads based on some already existing informations about design transactions in centralized, single-user design DBSs (e.g. DAMOKLES [DAM86b]).

The simulation model constructed relies on the reference architecture for design database systems presented in chapter 2. It models a server-workstation computer system where the server and the workstations communicate through a reliable communications network. Although our model can simulate cooperation between server and workstation (i.e. transfer of data between these nodes) at various levels of abstraction, we simulated only cooperation at the object level. That is, the server sends the workstation complete objects by CHECKOUT and receives from that node complete objects by CHECKIN.

The database system simulated supports two different transaction classes: conventional, short-duration batch transactions and long-duration design transactions. Batch transactions execute at the server node completely. Design transactions execute at workstations and spawn subtransactions on server to either check out objects of the public database or check in updated objects back in there. At the server node, batch transactions as well as CHECKOUT and CHECKIN operations are processed in the object/set-oriented buffer supported by the system´s object/tuple-oriented layer. Batch transactions are processed in the same way as in KARDAMOM: transaction updates are applied to selected tuple sets which are mapped back to database pages by transaction termination. CHECKOUT and CHECKIN operations are executed as in DAMOKLES. By CHECKOUT, the object´s main memory representation is constructed in the server´s object buffer and sent to the workstation. By CHECKIN, the object´s updated version is brought into the server´s object buffer and mapped onto database pages later on.

The server simulation model developed supports multi-level transaction management. The transaction managers of both system layers can lock either whole page clusters or single pages or subobjects or even complete objects. We used only page locks at both system levels so that page locks were kept on server for objects which were checked out by designers at workstations. By selecting the page as the lock granule, we assured correct execution of the algorithms which log at the page level. To further simplify the simulation model, cooperation between server and workstations is realized by means of a flat distributed transaction mechanism which relies on a two-phase commit protocol.

6.2.1 GM1´s Realization in the Simulation Model

6.2.1.1 Design Transactions at the Workstation

GM1 allows the designer at the workstation to check objects back into the (public) database at the server node at any time. Figure 6.1 depicts the dynamical properties of GM1 in the simulation model. Although the kernel system relates CHECKOUT locks to the design transaction, it releases these locks as soon as the respective objects are checked back into the public database.

GM1's design transaction (D-Tr) is started by the designer at the workstation. It consists of a sequence of object-oriented composed operations (OCOP) followed by a set of relational operations (ROP). The set of ROPs can be empty. In our performance study, design transactions executed only object-oriented operations while batch transactions processed relational tuples.

Each OCOP consists of a sequence of three recovery transactions (R-Tr). The first R-Tr starts a CHECKOUT operation at the server node. By this operation, a specific complex object (Oi) is copied from the public database into the private database at the workstation. By receiving D-Tr´s first CHECKOUT request, the public system creates a state record for the design transaction at the server node. After the CHECKOUT R-Tr has committed, the designer starts a PROCESS R-Tr. This recovery transaction processes Oi at the workstation's object-oriented buffer. After completely processing Oi, the designer starts a so-called CHECKIN R-Tr which is responsible for copying Oi back into the public database on server. By receiving a corresponding message from this transaction, the public system starts a short transaction (S-Tr) on server to actually check Oi into the database. If Oi has not been updated at the workstation, S-Tr only releases its locks; otherwise, S-Tr also maps Oi´s main memory representation onto database pages.

After all design transaction's OCOPs have been executed, the designer starts executing the set of relational operations. ROPs are also executed serially. Every ROP consists of only one recovery transaction. This R-Tr executes either a query or a short update operation directly in the public database. Since each query as well as each short update operation is completely executed in only one server call, there is no need to relate the locks which are acquired for these operations to the design transaction.

GM1's design transaction is considered to be terminated when all its OCOPs and all its ROPs have been executed. The DBS kernel blocks neither CHECKOUT operations nor ROPs which get involved in lock conflicts. The kernel simply aborts these operations and sends the respective workstation a message, instead.

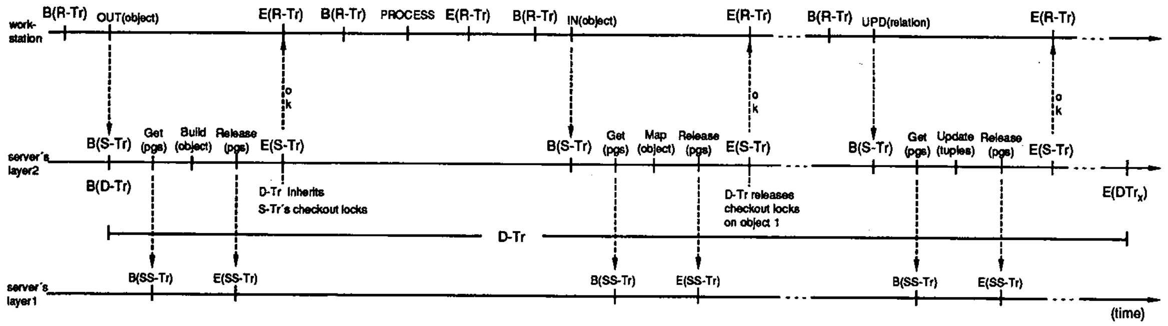
When the workstation receives an abort message, the designer starts either another OCOP or another ROP. He tries to restart the aborted operation only when all other operations have already been executed. In the case there are no more operations to be executed, the designer keeps on restarting the aborted operation until it can be executed at the server node.

6.2.1.2 Remote Operations and Batch Transactions at the Server Node

Until now, we have only explained design transaction execution at the workstation in more details. In the following, we explain data processing activities at the server node. Every remote operation is processed by the kernel system as a short transaction (S-Tr). Short transactions execute at the server´s object/tupe level and can start subtransactions (SS-Tr) at the server´s page/segment level to read or write database pages. Design transactions inherit CHECKOUT locks from those S-Trs which execute CHECKOUT operations on their behalf. The DBS kernel releases CHECKOUT locks in the public database when the respective CHECKIN S-Trs commit.

As already explained in the last section, the kernel does not only execute remote operations of design transactions. It also processes batch transactions. Batch transactions are directly started at the server node. Everyone of these transactions consists of a sequence of (short) relational operations. These operations are similar to the ROPs of design transactions. The kernel executes relational operations of batch transactions as short transactions (i.e. at its object/tuple level). At the end of a relational operation, the kernel checks if there is a next operation to be executed on behalf of the batch transaction. When all operations of the batch transaction have already been executed, the kernel commits it. Depending on the commit protocol being used (which, in turn, depends on the recovery algorithm being tested), the server can map relational updates which are processed in the object buffer onto database pages either after every batch transaction´s operation or only at transaction commit. In any case, though, relational updates must be saved at the end of each relational operation.

Contrary to object-oriented operations and ROPs of design transactions, batch transactions can be blocked by the public system when some of their relational operations participates in a lock conflict. Deadlock prevention is realized at the server node by means of a timeout mechanism. After being blocked for a previously specified period of time, batch transactions are aborted and brought back into the server´s ready queue for restart.



B(Tr): begin transaction Tr; E(Tr): terminate transaction Tr; D-Tr: user's design effort at the server; R-Tr: recovery trans. at the workstation; S-Tr: short trans. at the server's tuple/object level; SS-Tr: subtrans. of a short trans. at the server's page level;

Fig. 6.1: GM1's implementation in the simulation model

6.2.2 The Architecture of the Server's Simulation Network

The server network was realized in two layers. The lower (physical) layer simulates hardware as well as operating system functions. These functions are realized by the host machine on top of which the DBS kernel (i.e. the public system) is installed. The higher (logical) layer of the simulation model represents the whole design environment (i.e. the kernel DBS, the communications network, and the private systems at the workstations). In the following, we describe each network layer. Then, we present the run-time parameters which control network simulation.

6.2.2.1 The Network's Logical Layer

Figure 6.2 shows the logical layer of the simulation network. In this figure, boxes represent simulation nodes and arrows model communication between nodes. Simulation nodes represent software modules that process transactions. The network's logical layer can further be subdivided into three (sub)networks: a network representing the workstations, another one for the communications subsystem, and a third network representing the kernel software at the server processing node.

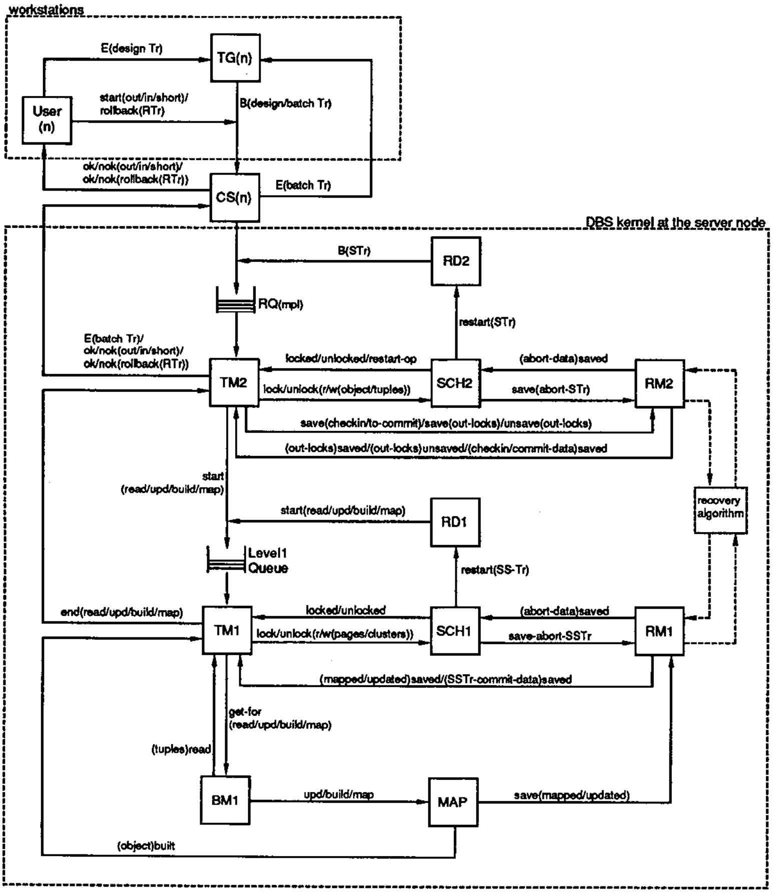
Each simulation node implements a set of operations. These operations form the node's interface. In Figure 6.2, the interface of each simulation node is represented by the operations which identify the arrows pointing to the node. Transactions are passed from node to node until they are completely processed. Each simulation node processes transactions sequentially and maintains an internal queue where incoming transactions are kept until they can be processed by the node.

Each simulation node is identified by a specific mnemonic associated with it. Thus, **TG** stands for transaction generator, **User** for user node, **CS** for communications subsystem, **TM** for transaction manager, **SCH** for scheduler, **RM** for recovery manager, **RD** for restart delay, **BM** for buffer manager, and **MAP** for mapping module. Besides simulation nodes and communication arrows, the simulation networks's logical layer presents two external queues: the ready queue (**RQ**) and the Level1-Queue.

We suppose the reader is familiar with the functions of most of the software modules cited above. Perhaps the only nodes which deserve a more detailed explanation are both **User** and **MAP**. The user node models the work of designers at workstations. This node simulates the execution of PROCESS recovery transactions. Moreover, the user node determines the next operation to be started on D-Tr's behalf and controls the end of design transactions. PROCESS recovery transactions are simulated in the user node by means of time delays.

MAP simulates the operations which build the main memory representation of objects and tuple sets out of sets of database records and tuples, respectively. Moreover, **MAP** is also responsible for mapping transaction results in main memory representation onto database pages. Costs of **MAP** operations are expressed in number of CPU instructions.

Following the reference architecture of chapter 2, the simulation network representing the kernel software can further be subdivided into two (sub)networks, each one of them representing one of the server's software layers (i.e. L1 and L2 in Figure 1.4). The network which is built by the ready queue and the nodes **TM2, SCH2, RD2,** and **RM2** models the server's object/tuple-oriented layer (L2). The rest of the nodes which simulate kernel software modules together with the Level1-Queue model the server's page/segment-oriented layer (L1).


workstations
E(design Tr)
TG(n)
User (n)
start(out/in/short)/ rollback(RTr)
B(design/batch Tr)
ok/nok(out/in/short)/ ok/nok(rollback(RTr))
CS(n)
E(batch Tr)
DBS kernel at the server node
B(STr)
RD2
RQ(mpl)
restart(STr)
E(batch Tr)/ ok/nok(out/in/short)/ ok/nok(rollback(RTr))
TM2
locked/unlocked/restart-op
lock/unlock(r/w(object/tuples))
SCH2
(abort-data)saved
save(abort-STr)
RM2
save(checkin/to-commit)/save(out-locks)/unsave(out-locks)
(out-locks)saved/(out-locks)unsaved/(checkin/commit-data)saved
start (read/upd/build/map)
start(read/upd/build/map)
RD1
recovery algorithm
Level1 Queue
restart(SS-Tr)
end(read/upd/build/map)
TM1
locked/unlocked
lock/unlock(r/w(pages/clusters))
SCH1
(abort-data)saved
save-abort-SSTr
RM1
(mapped/updated)saved/(SSTr-commit-data)saved
get-for (read/upd/build/map)
(tuples)read
BM1
upd/build/map
MAP
save(mapped/updated)
(object)built

Note (1): the meta symbol "/" is used to separate alternative messages as well as operations and parameters; Note (2): TG(n) stands for transaction generator and indicates that n workstations are active in the system; User represents the user work at the workstation; CS stands for communications subsystem; RQ represents the server's ready queue; the indices on the other nodes indicate whether they are located in the server's lower layer (1) or in its upper layer (2); TM stands for transaction manager; SCH stands for scheduler; RD represents a delay before the restart of some transaction; RM stands for recovery manager; BM stands for buffer manager; MAP represents the module which changes the object's representation.

Fig. 6.2: The logical layer of the simulation network

The box for the recovery algorithm represents no simulation node. Recovery activity actually is simulated by both **RM1** and **RM2**. We designed the box for the recovery algorithm only to make clear that **RM2** and **RM1** can cooperate with each other by the implementation of the recovery algorithm. Moreover, depending on the recovery technique being simulated, either **RM1** or **RM2** can even become unnecessary.

Besides being activated by other simulation nodes, recovery nodes can be directly activated by the simulation supervisor. By means of the crash delay run-time parameter (*Dcrash*) the time interval between any two consecutive system crashes can be passed to the supervisor. The simulation system automatically controls this delay and informs the recovery nodes when a system crash occurs. Instead of using delays to control checkpoint generation, checkpoints were made dependent on the log file´s size. That is, the recovery nodes generate a new checkpoint every time the log becomes full. The log file´s size is passed to the simulation system by means of a run-time parameter (*Log-Size*).

No object-oriented buffer manager has been modeled in the simulation network. It was assumed that object buffer size in both server and workstation is large enough so that no virtual memory management for the object buffer is necessary in either of the two processing nodes.

Transaction load is generated off-line. **TG** only reads transactions from the load file, introduces them in the simulation network (begin-transaction operation), and receives committed transactions from other nodes. By controlling both the number of committed transactions and the end of the load file, **TG** can identify the end of the simulation. Figure 6.2 associates **TG** with an index in parentheses (**n**). It represents the number of active users being simulated. At any time during the simulation, each of these **n** users is executing one transaction (of any class).

Both scheduler nodes (i.e. **SCH2** and **SCH1**) realize locking mechanisms to synchronize concurrent transactions. Transactions can access data in one of three modes: read mode (r), write mode (w), and read-intention-to-write mode (riw). The use of the riw lock mode depends on the state of a specific simulation parameter (*Lock-Upgrade*). If this parameter is set to TRUE, the data which will be updated later on are first locked in riw mode; otherwise, these data are directly locked in write mode. riw locks are upgraded to write locks only when the transaction decides to update the data at the server. Figure 6.3 presents the lock compatibility matrix for the set of lock modes associated with the simulation network.

	r	riw_b	riw_d	w
r_b	ok	-	ok*	-
r_d	ok	-	-	-
riw	ok	-	-	-
w	-	-	-	-

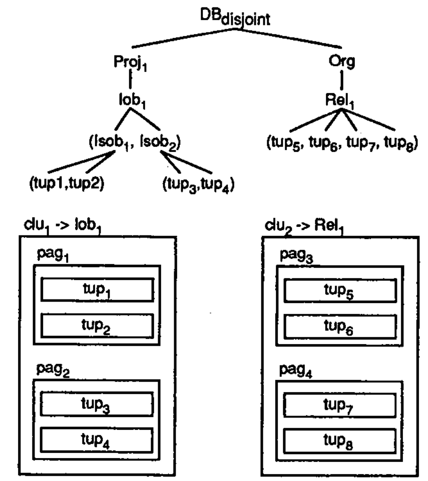
Note: r stands for read mode, w for write mode, and riw for read-intention-to-write mode; the indices d and b stand for design transaction and batch transaction, respectively; the locks on the figure left side are being requested.
* only if the corresponding write lock has not yet been requested

Fig. 6.3: Lock compatibility matrix for both SCH2 and SCH1

We complement our description of the kernel network's logical layer with an example. By this example, we make the following assumptions. The kernel software implements a multi-level transaction manager at both, L2 and L1. In L2, short transactions (S-Tr) are synchronized on the basis of tuple locks. L1's (sub)transaction manager realizes a page locking mechanism.

Figure 6.4 shows a design database example as well as the record of a design transaction (D-Tr), and the record of a batch transaction (B-Tr). D-Tr checks a complex object (Iob_1) out, processes it at the workstation, and checks it back into the public database later on. After that, D-Tr reads one relational tuple ($Rel_1.tup_5$). B-Tr which we suppose begins when D-Tr starts reading $Rel_1.tup_5$ reads two tuples ($Rel_1.tup_5$ and $Rel_1.tup_6$) and updates one of them.

TG reads D-Tr's record (DR) from the load file, sets its actual state to *start-transaction*, and sends it to the communications subsystem. **CS** identifies DR's actual destination. It must be sent to the server's ready queue. Before doing that, **CS** stores DR in an internal queue, though. After a simulation time delay which represents the communication delay, **CS** sends DR to **RQ**. The ready queue controls the multiprogramming level (mpl) of the kernel system. mpl's value is determined by a run-time parameter (*mpl*) at the beginning of the simulation run. DR is kept in **RQ** until the number of transactions running in the public system becomes smaller than *mpl*. At this moment, DR is sent to **TM2**.



D-Tr: {out(Iob_1), process(Iob_1), in($Iob_1.Isob_1$), read($Rel_1.tup_5$)}

B-Tr: {read($Rel_1.tup_5$, $Rel_1.tup_6$), write($Rel_1.tup_5$)}

Fig. 6.4: Data structures and transactions for the simulation example

TM2 notices that DR just began, identifies its first operation, associates DR with a CHECKOUT S-Tr, sets DR's state to *read-lock*, and sends DR to **SCH2**. This simulation node, then, tries to lock the tuples belonging to the object being checked out by D-Tr (i.e..tup_1, tup_2, tup_3, and tup_4 in cluster clu_1). If any of these tuples is already locked, **SCH2** changes DR's state to *restart-op* and sends DR back to the transaction manager; otherwise, the scheduler grants CHECKOUT S-Tr the requested tuple locks, sets DR's state to *read-locked*, and sends DR back to **TM2**. From this node, DR is sent to the server's lower layer. It enters this layer through the Level1-Queue and is analyzed by **TM1**. This node identifies DR's actual state (*start-build*), envelops DR in an SS-Tr, and sends it to **SCH1**. The scheduler locks pag_1 and pag_2, changes DR's state to *read-locked*, and sends DR back to the transaction manager. This node sends DR further to the buffer manager. **BM1** searches for the requested pages in the page/cluster-oriented buffer. In case clu_1 is not present in the buffer, **BM1** starts an I/O-operation to copy this cluster from disk into the buffer. After making the requested cluster accessible in the buffer, **BM1** sets DR's state to *build* and sends DR to the mapping simulation node. **MAP** calculates how many CPU instructions are needed to build the object-oriented representation of Iob_1, changes DR's state to *object-built* and sends DR back to **TM1** (via the CPU simulation node of the network's physical layer). DR's state, then, is set to *unlock-pages* and the record is given the scheduler. **SCH1** releases both pag_1's and pag_2's locks, changes DR's state to *unlocked*, and sends DR back to **TM1**. This transaction manager sends DR to **TM2**, after committing the SS-Tr related to it.

Before being sent back to the user node via communications subsystem, DR is processed by **RM2**. This node associates S-Tr's tuple locks with D-Tr and saves them as CHECKOUT locks (possibly in a special log file). Iob_1 is processed in the user node further. **User** calculates the processing time for this object on the basis of the number of records belonging to it. DR is kept in the user node during this time interval. At the end of the process time, **User** sets DR's state to *start-in* and sends DR back to **TM2** via **CS** and ready queue. **TM2** creates a CHECKIN S-Tr and relates it to DR. After that the transaction manager changes DR's state to *save-checkin* and sends DR to the recovery manager. If the recovery technique being simulated saves object updates before they have been mapped onto database pages, this action is executed by **RM2** at this point in time; otherwise, the recovery manager simply sets DR's state to *checkin-saved* and sends DR back to **TM2**. DR is further sent to the server's page/segment layer where Iob_1's updates are mapped onto pag_1 and pag_2. At the end of both the data mapping and the possible data saving operations at L1, DR is sent back to L2. Then, **SCH2** releases D-Tr's tuple locks and sends it back to **RM2** via the transaction manager. **RM2** invalidates D-Tr's CHECKOUT locks and sends DR back to **TM2**. This node commits the CHECKIN S-Tr and sends DR to the user node.

User searches the next operation to be executed on D-Tr's behalf. It is a query. tup_5 must be read in the public database. **User** sets DR's state to *start-short* and sends DR back to **TM2**. The transaction manager starts a new S-Tr for DR and sends it to the scheduler. **SCH2** locks tup_5 and sends DR to the server's lower layer via **TM2**. We can imagine that **TG** introduces B-Tr in the simulation network in parallel to the above operations. When **SCH2** tries to lock both tup_5 and tup_6 on behalf of B-Tr, it identifies the lock conflict with DR's S-Tr. B-Tr, then, is blocked during the execution of DR's S-Tr. That is, B-Tr's transaction record (BR) is kept in **SCH2**'s block list until D-Tr's tuple locks are released in the public database.

When the scheduler releases the locks held by DR's S-Tr, it unblocks B-Tr, grants it the locks for tup_5 and tup_6, changes the state of BR to *read-locked*, and sends it back to the transaction manager which, in turn, sends BR to the server's lower layer. **TM2** receives BR back, after tup_5 and tup_6 have already been read out of pag_3 and copied into a tuple set for further processing. **TM2** changes BR's state to *write-lock* and sends BR to the scheduler. **SCH2** upgrades the lock mode of the locks held by B-Tr and sends BR back to the server's page/segment layer via transaction manager. If the recovery technique being simulated saves updated tuples, before they are mapped onto pages, **RM2** executes this activity before BR is sent to **TM1**; otherwise **RM1** saves data updates, after the mapping node has copied tup_5's new version onto pag_3. After doing that, the recovery manager sends BR back to **TM1**. The transaction manager notices that the updated tuples have just been saved and sends BR to the scheduler. **SCH1** releases B-Tr's page lock and sends BR back to the server's tuple/object layer via **TM1**. After B-Tr's tuple locks have been released by **SCH2**, **TM2** commits B-Tr and sends it´s BR back to TG via **CS**.

6.2.2.2 The Physical Layer of the Kernel Network

To simulate finite computing resources at the server processing node, we mapped those simulation nodes representing kernel software modules (e.g. **TM2**) onto another simulation network that models CPU and disk resources. This network represents the physical layer of the server processing node. It´s layout is sketched in Figure 6.5. Besides **RD2** and **RD1**, all simulation nodes modeling kernel software are treated by the physical layer as operating system processes which consume services of both CPU and disk units. The idea of modeling the server node as a hierarchy of simulation networks is based on the simulation models presented in [Care83], [CaSt84], and [AgCL87] for the evaluation of concurrency control algorithms in centralized database systems.

Figure 6.5 shows how process execution is modeled by the kernel´s physical layer. Idle logical nodes, that is, simulation nodes of the logical layer which are processing no transaction records (TR) at the moment wait in the physical layer's idle node. When a simulation node SN receives a TR, it moves into the CPU´s input queue. By entering the CPU, SN processes TR. This processing activity is simulated through CPU time delays. Since each process receives only a finite slice of the CPU time (which is measured in number of CPU instructions), it is possible that SN must re-enter the CPU more than once until it can completely process TR. If SN concludes that I/O-operations must be executed on TR´s behalf, it first identifies the disk on which the needed data are stored and, then, moves into the input queue of the corresponding disk unit. Every I/O-operation consists of a data transfer operation and a sequence of operating system operations. Thus, SN must come back to the CPU, after the data transfer operation has been executed. After completely processing TR, SN tests if there is another transaction record to be processed. If no new TR is waiting in SN´s input queue at the kernel´s logical layer, SN moves back into the idle node of the physical layer; otherwise, SN moves into the CPU´s input queue and starts processing the next transaction record. The simulation supervisor program awakes logical nodes waiting in the idle node of the physical layer when some new TR is brought into their input queues.

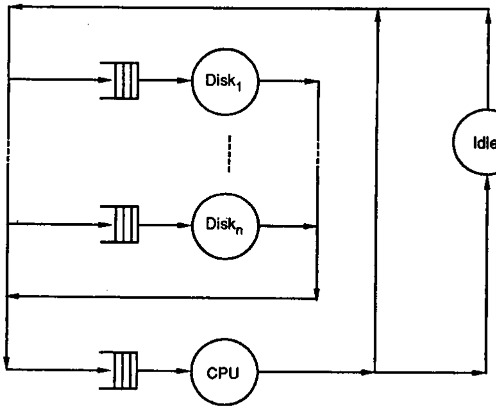


Fig. 6.5: The physical layer of the simulation network

6.2.2.3 Simulation Network's Run-Time Parameters

As it can be derived from the explanations above, specific characteristics of different simulation runs can be established through a set of run-time parameters. This set can be subdivided into three different groups of parameters. The first group expresses costs as time delays. These parameters are mainly used in modeling I/O-operations, communication delays, user processing times, block and timeout delays, etc. The second group represents those parameters which express costs in quantities of CPU instructions. Most parameters of this type model costs for logical simulation nodes. The third group of parameters model the system architecture itself. The parameter establishing the server's multiprogramming level and the one expressing the number of active users (terminals/workstations) in the system are examples of parameters belonging to this third group.

Figure 6.6 shows a list of some important run-time parameters of the simulation network with possible values associated with them. **CPUbuild/map** fixes the number of instructions per data record necessary to build/map a design object. Similarly, **CPUread/write** establishes the number of instructions per tuple necessary to build/map a tuple set. **CPUrec** fixes the costs for processing a data record or tuple in main memory. **Duser** determines how long the designer processes each data record (of a complex object) at the workstation. **Dio** establishes the duration of I/O-operations on database disks. **Dlogio** fixes the time necessary to either read data from or write data to the log file (on the log disk). If **Chained-I/O** is set to TRUE, every I/O-operation can transfer up to one cluster of data pages from/to disk. If this parameter is set to FALSE, instead, every I/O-operation can transfer only one page from/to disk. As the name implies, **Buffer-Size** fixes the size of the server's page/segment-oriented buffer. Finally **CPU-Speed** indicates how many CPU instructions are executed per simulated microsecond.

CPUbuild/map	300000 instr	Duser	1800 ms/tuple	Chained-IO	TRUE
CPUread/write	25000 instr	Dio	30 microseconds	Buffer-Size	4 Mbytes
CPUrec	20 instr	Dlogio	15 microseconds	CPU-Speed	10 MIPS

Fig. 6.6: Possible values of some simulation parameters

6.3 Describing the Simulation Scenarios

As already explained in section 6.1, we investigated the cost of different recovery activities separately. To do that, four basic simulation scenarios were defined: the normal system operation, the transaction abort, the system checkpoint, and the system crash environments. Transactions were forced to back out in the transaction abort environment by reducing the allowed transaction blocking time. After the blocking time had elapsed, transactions were forced to abort due to the timeout mechanism realized by the public system to prevent the occurrence of deadlocks. Since only business-oriented transactions could be blocked at the server node, they were the only ones which aborted by the simulation of the transaction abort environment. Checkpoints were forced in the system checkpoint environment by limiting the maximum size of the log file on disk. Depending on the recovery mechanism being simulated in that environment, the maximum log size was achieved sooner or later. The volume of information kept on stable storage by each of the simulated recovery mechanisms was analyzed on the basis of the normal system operation environment for which no log size limit was set. The number of system crashes which should occur during simulation was set by means of a run-time parameter which expresses the time interval between two consecutive crashes. Since crash recovery took longer by some mechanisms and no system crash was allowed to occur during crash recovery, the number of crashes was not always the same by every simulation of the system crash environment.

We also wanted to investigate recovery performance under different transaction load characteristics. Similar to the performance evaluation in [Reut84], we were interested in analyzing recovery performance in processing environments which present either a high or a low frequency of update transactions, since recovery activity is closely related to the update activity in the database system. Therefore, three load parameters were varied as to produce four different transaction load types. These parameters respectively determine the number of design transactions (as well as the number of business-oriented transactions) in the load, the number of update transactions, and how much of the read data is actually updated by update transactions. We present the four basic transaction load types below.

- *Load1:* High update transaction rate (80%) and high volume of updated data (80% of the transaction´s read data) combined with high design transaction rate (33%).
- *Load2:* High update transaction rate and high volume of updated data combined with low design transaction rate (about 10%).
- *Load3:* Low update transaction rate (30%) and low volume of updated data (51% of the transaction´s read data) combined with high design transaction rate.
- *Load4:* Low update transaction rate and low volume of updated data combined with low design transaction rate.

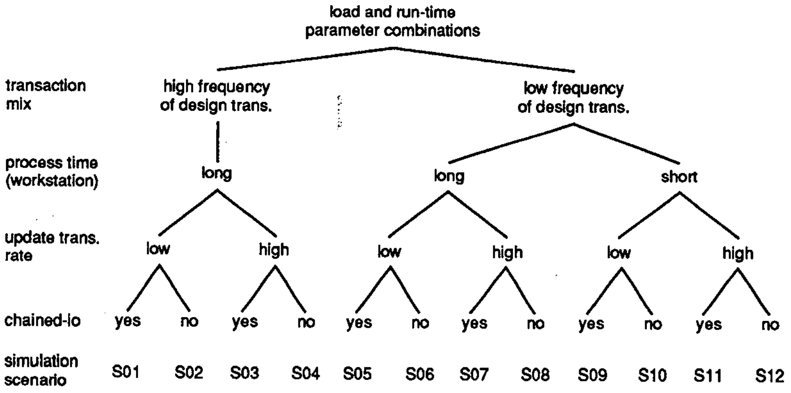


Fig. 6.7: Classification of the basic simulation scenarios

Although 33% seems not to represent a high design transaction rate in the overall transaction load, it should be considered that each design transaction starts 6 short transactions at the server node (i.e. 3 CHECKOUT and 3 CHECKIN operations) while each business-oriented (batch) transaction represents only one S-Tr in the public system. Load files of type Load1, for instance, force the server node to execute 1851 CHECKOUT S-Trs, 1851 CHECKIN S-Trs, 300 read-only batch transactions, and 1083 update batch transactions. Each transaction load file generated for simulation relies on one of the load types described above and consists of 2000 transactions.

To simulate different operating system environments as well as different designer behaviors, we also varied the values of some run-time simulation parameters. By varying the processing time at the workstation (i.e. the duration of the PROCESS R-Tr) and maintaining the multiprogramming level at the server node constant, we controlled the number of active workstations and terminals which were simulated. This number varied between 10 and 87. On the other hand, we simulated operating systems which support chained-I/O and those which supported only page-oriented I/O-operations, because we were interested in investigating how chained-I/O affects the performance of page-oriented logging mechanisms. By combining the values of different run-time simulation parameters, four basic run-time parameter settings were produced:

- *Setting 1*: Long-duration PROCESS R-Tr (i.e. 3 minutes per object at the workstation) combined with chained-I/O.
- *Setting 2:* Long-duration PROCESS R-Tr combined with simple I/O-operations.
- *Setting 3:* Short-duration PROCESS R-Tr (i.e. 1 minute per object at the workstation) combined with chained-I/O.
- *Setting 4:* Short-duration PROCESS R-Tr combined with simple I/O-operations.

By most of the simulation runs, the values of all other run-time parameters were made dependent of the setting type selected. All combinations of run-time as well as load parameters were calculated to produce simulation runs where transaction processing activities consume some 85% of the system resourçes (i.e. CPU and disks). The remainder 15% of computer resources were supposed to be consumed by other operating system´s activities (e.g. garbage collection). CPU capacity on server was usually kept by 9 MIPS. By the simulation of some extra scenarios, we increased CPU capacity (e.g. 15 as well as 25 MIPS) to change the proportion between DBS CPU time and operating system CPU time. In this way CPU idle time was increased and more parallelism between user transactions and deferred mapping operations was achieved.

By combining different load types with the various run-time parameter settings, we obtained 12 different simulation scenarios. Figure 6.7 depicts the main characteristics of these scenarios. We did not consider the combination of high design transaction rate with short-duration PROCESS R-Tr, because this scenario seemed to be similar to the one where low design transaction rate is combined with long-duration PROCESS R-Tr. By combining these twelve scenarios with the four simulation environments presented in section 6.1, we produced a set of 48 basic scenarios. 240 simulation runs were necessary to investigate the behavior of the five recovery algorithms in these 48 different processing scenarios. On the basis of the results obtained through these set of simulation runs, we decided to investigate the performance of the recovery mechanisms in a number of extra scenarios where the values of some other (load as well as run-time) parameters were varied independently. All extra scenarios which were simulated were based on scenario S03. We were very interested in analyzing recovery performance for this scenario in more detail, because it combines high design transaction rate and high update transaction rate with long-duration PROCESS R-Tr and chained-I/O. We believe that most of the future integrated information systems will present these properties, too. The following extra scenarios were created to complement our simulation study:

- *SE1:* The scenario S03CH was simulated with a transaction load consisting of 2000 design transactions (i.e. design transaction rate = 1.0).
- *SE2:* In this scenario, the locality of access presented by design transactions as well as batch transactions was raised from 10% to 30%.
- *SE3:* To investigate recovery performance in processing environments where mapping operations take not so long, we produced a scenario where these operations have a cost of 128000 machine instructions per database record instead of the original 384000 instructions. The original cost of representation mapping operations which was applied to all other simulation scenarios is based on the cost of these operations in the DAMOKLES database system.
- *SE4:* By the simulation of this scenario, the server´s CPU capacity was set to 11 MIPS. As already explained above, we wanted to investigate the performance of recovery mechanisms which support deferred mapping in processing environments with higher CPU capacity. Actually, because of the high costs associated with

CHECKOUT and CHECKIN operations, database systems supporting integrated information systems will have to rely on servers with much higher processing capacities than the 9 MIPS simulated in the other scenarios. This will not be difficult even for microprocessor-based, centralized server processing nodes, though. The 32-bits microprocessor Motorola M68040, for instance, should be brought to the market at the end of 1989 and will be able to process some 24 million instructions per second. Through the simulation of scenario SE4, we investigated the behavior of recovery mechanisms by a little increase on the CPU capacity (i.e. some 20%). Other scenarios were designed for the analysis of recovery performance under higher CPU capacities (e.g. SE5 and SE9).

- *SE5:* In this scenario, the server's processing capacity was set to 25 MIPS. We simulated this scenario for the same reasons which led us to simulate SE4.
- *SE6:* This as well as the next extra scenario to be presented were used to investigate the effects of deferred mapping in a design cooperation environment. In this scenario, we supposed that 20% of the CHECKIN operations executed at the server node bring non-committed design object versions into the group database. These object versions, then, are soon checked out by other designers.
- *SE7:* This scenario is similar to SE6 but CPU capacity is set to 25 MIPS here.
- *SE8:* In this scenario the maximum log size was set to the half of its normal value. On the basis of this scenario, we simulated S03 combined with the system crash environment (CR) to investigate how a smaller ring file can affect crash recovery time for REC2 and REC5.
- *SE9:* This scenario is based on S03CH but presents a CPU capacity of 50 MIPS.
- *SE10:* This scenario is based on SE6 but presents a CPU capacity of 11 MIPS.
- *SE11:* This scenario is based on SE1 but presents a CPU capacity of 15 MIPS.

6.4 Main Results of the Recovery Performance Evaluation

In the present section, we first present some of those evaluation results of the simulation analysis of recovery algorithms that can be generalized for all (or almost all) processing scenarios simulated. Then, we describe selected simulation results which apply either to specific recovery techniques or to special simulated scenarios. The whole set of simulation results classified by simulation scenario can be found in [Ioc89b].

6.4.1 Recovery Cost in the Design Environment

Recovery has proven not to be the most resource consuming component of the public system. This was true for all recovery mechanisms simulated. System components such as the buffer manager or the mapping module consumed a lot more CPU time than the recovery modules. This observation was confirmed even by the simulation of scenarios with a high update transaction rate. For instance, by the simulation of REC1 in scenario S01CH (i.e. S01 with checkpoint generation) **RM1** was visited 4918 times by batch transactions. On the average, by every visit the transaction waited about 10.5ms to be processed by **RM1** and was processed in some 0.13ms. In the same simulation run, transactions waited about 102.8ms in **BM1**'s internal queue and were processed by **MAP** in some 512ms. Design operations (CHECKOUT and CHECKIN) waited even longer. While each design operation was kept in **RM1**'s queue for almost the same time as batch transactions were, it waited 149ms in **BM1**'s queue on average and was processed by **MAP** in about 8.5 seconds.

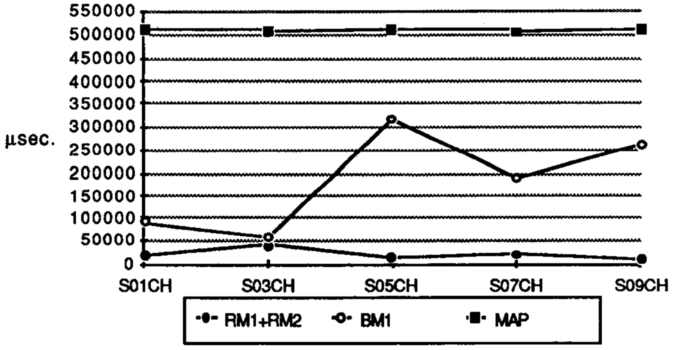


Fig. 6.8: Costs in microseconds for each visit of a transaction to either the recovery, buffer manager, or mapping node

Figure 6.8 shows three curves of costs. They depict the costs of recovery activities, buffer management, and mapping operations in various simulation scenarios, respectively. These costs are related to batch transactions only. They represent the average time a transaction spends at the respective simulation node by each visit to this node. The average time includes both the time during which the transaction waits to be served and the service time itself. It is easy to note that the mapping time is almost constant for all simulation scenarios. This can be explained by the fact that the objects read as well as those written by the transactions, respectively, have the same size for all simulation scenarios. Both BM1 and recovery costs depend on the scenario being simulated, though. Buffer management costs are higher in scenarios with high read-only batch transaction rates. In these scenarios, data is read but not updated. Therefore, the number of accesses to the same data in the page-oriented system buffer is reduced and the probability with which next data accesses will require I/O-operations increase. Recovery activities reach their highest costs by the simulation of S03CH. This scenario presents the highest design transaction rate combined with high update transaction rate. By the simulation of all scenarios shown in Figure 6.8, though, recovery costs were always significantly lower than the costs of other system activities. In principle this can be explained by the high costs of mapping operations and buffer management. On the other hand, waiting times were very high in the simulation network, because of the relative low CPU capacity of the server (i.e. 9 MIPS).

6.4.2 The Effects of Different Transaction Classes on the Simulation Results

The simulation study has shown that transaction execution can take much longer in integrated database system environments which support very different transaction classes. Growing response times could be observed especially by the execution of batch transactions. The ones simulated in our study take about 3 seconds when they execute alone on server. Simulated together with design transactions, these batch transactions took up to 4 minutes to be completely processed (this occurred, for instance, by the simulation of update batch transactions in S03CH).

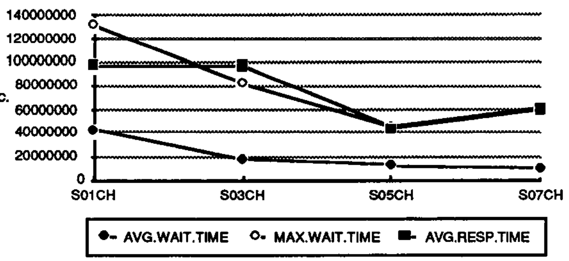


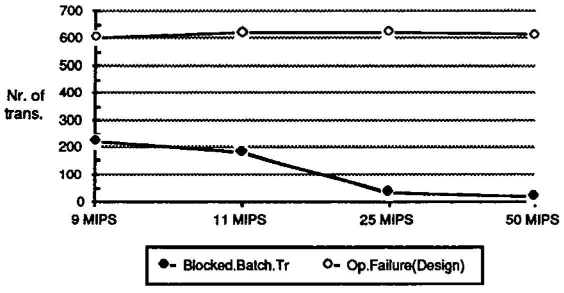
Fig. 6.9: Batch transactions' average and maximum times in the server's ready queue

Batch transactions spent great part of their execution time waiting in the ready queue of the server node. Since the multiprogramming level at the public system was kept low (e.g. up to 10 transactions) and both CHECKOUT as well as CHECKIN operations practically consumed all system resources, starting batch transactions were often kept in the ready queue for long periods of time, before they were allowed to really enter the system. The graphic in Figure 6.9 shows both maximum and average times spent by batch transactions in the server's ready queue during the simulation of various scenarios. Moreover, the graphic also shows the curve for batch transaction average response time in those scenarios. The time spent by batch transactions in the ready queue mostly depended on the design transaction rate as well as on the update transaction rate, and the number of terminals being simulated. By the simulation of S01CH, for instance, 43% of the response time was wasted in the ready queue on average. The maximum time in that queue even exceeds the average response time for batch transactions in S01CH. This can mainly be explained by the high number of both long-duration CHECKOUT operations and active terminals (38) in this scenario. By the simulation of S03CH, the waiting time in the ready queue dropped to some 17% of the total response time. This reduction on the waiting time was only in part caused by the reduction on the number of simulated workstations (25). Another important cause of it is the fact that the number of CHECKOUT operations which were aborted by the server increased. Since the update transaction rate in S03CH is higher than the one in S01CH, design transactions tend to retain acquired CHECKOUT locks for longer periods of time. As a consequence, the number of new CHECKOUT operations which are rejected because of access conflicts is higher in S03CH (601) than in S01CH (126). Waiting batch transactions are brought into the public system faster when running CHECKOUT operations terminate early. Finally, in S05CH the average time spent by batch transactions in the ready queue represents 28.5% of their total response time. Although this time is not shorter for S07CH, it represents only 15.4% of the response time in that scenario. This can be explained by the high update transaction rate of S07CH's transaction load. That is, since most transactions in S07CH update the data they read, they usually take longer than transactions which execute in S05CH, since this scenario presents a low update transaction rate.

6.4.3 Relative Processing Capacity of the Server Node

In scenarios which presented both high design transaction and high update transaction rates, REC2 reduced response time for update batch transactions by 25%. On the other hand, it increased response time for CHECKOUT by 15% in these scenarios. Only by increasing CPU capacity, it was possible to reduce response time for all transaction classes at the same time. By increased processing capacity, the server could execute transactions faster. This had two consequences. First, running remote design operations (e.g. CHECKOUT) did not consume almost all server resources anymore. Moreover, since remote operations were processed faster, batch transactions did not have to wait in the ready queue for so long. On the other hand, by higher CPU capacity, the public system could reduce the thrashing effect among batch transactions. That is, the number of blocked (and also aborted) transactions was reduced on server, since access conflict times became shorter. This last effect of increased CPU capacity is depicted in the graphic of figure 6.10. The lower curve shows how the number of blocked (batch) transactions decreases as CPU capacity increases. The higher curve in this graphic shows, on the other hand, that the number of CHECKOUT operations which fail in the public system due to access conflicts practically remains constant while processing capacity increases. This can be explained by the fact that access conflicts among design transactions are of longer duration and also depend on the time during which design objects are processed at the workstation.

In most of the simulation experiments realized, the server's CPU capacity was set to 9 MIPS and the transaction load simulated was designed to consume 85% of the server's processing capacity with a multiprogramming level of 10 transactions. Simulation results showed that operating system's activity (e.g. queue management) consumed much more than the 15% of CPU capacity reserved for them. As a consequence of that, thrashing as well as waiting times in the ready queue increased. This characteristic of some simulation runs made it difficult to identify small differences in the behavior of the recovery mechanisms investigated. Therefore, we decided to simulate some scenarios on the basis of more powerful server processing nodes. Later in this section, we compare the results obtained by the simulation of the same scenarios using different CPU capacities.



Note: these results were obtained by the simulation of REC1

Fig. 6.10: Relating the number of blocked and aborted transactions to CPU capacity

6.4.4 Recovery Performance on the Basis of Different Simulation Scenarios

In the following, we rely on the classification of simulation scenarios shown in Figure 6.7 to present some general results of our recovery evaluation. Mainly as a consequence of both the long waiting times spent by transactions in the server's ready queue and the long-duration mapping operations executed by the server, recovery activities did not influence system throughput very much even when CPU capacity was increased. Furthermore, the different recovery mechanisms affected system throughput in a similar way. They showed much more differences in what concerned transaction response time. Figure 6.11 relates system throughput with recovery for various checkpoint-oriented scenarios when CPU capacity is kept by 9 MIPS. These scenarios respectively present different design transaction rates. It is easy to note that the recovery strategy has practically no influence on the system throughput during normal system operation. By scenarios where transactions may be rolled back, different recovery mechanisms can affect system throughput differently, though. Figure 6.12 relates throughput in some transaction-abort scenarios with the recovery techniques simulated. While throughput almost remains constant for S03CH, it increases by the simulation of REC2 in both S07AB and S11AB. By most of the simulated scenarios, REC2 has reduced the response time of update batch transactions while increasing the processing time of other transactions. Since the former transactions respectively represent 53.1% and 54.5% of the transaction loads associated to S07AB and S11AB, REC2 allowed a little increase of system throughput in these scenarios.

It is interesting to note how the processing time at the workstation can affect the performance of REC5 by low CPU capacity at the server node. Figure 6.12 shows that while REC5's performance is similar to that of REC3 and REC4 in scenario S07AB, it relatively decreases in scenario S11AB. By committing design transactions sooner and allowing new transactions to start in parallel to mapping operations, REC5 increases the processing load on server. This load becomes even greater when objects which are transferred to the workstation return to the server sooner. This is exactly what happens by the simulation of S11AB. Consequently, the number of access conflicts in the public database increases and more transactions must be either blocked or aborted.

Although recovery mechanisms based on deferred mapping can help to increase system throughput in environments with high abort transaction rates, they cannot compete with other recovery techniques in environments where system crashes occur frequently. These mechanisms recover the system state after a crash by reading updated design objects and tuple sets from the log and forcing them to be mapped on database pages. Since much of the logged data had already been mapped once before the crash occurred, these new mapping operations represent an extra burden to the database system. Figure 6.13 shows how transaction throughput is affected by the various recovery algorithms when different system-crash scenarios are simulated. An interesting observation related to this figure is that REC4's performance does not differ from that of REC1 and REC3 very much. Although these two other mechanisms execute more complex crash recovery algorithms than REC4, these algorithms must be processed relative seldom. On the other hand, most of the I/O-operations realized by REC1 and REC3 during crash-recovery activities are executed by REC4 during transaction commit processing.

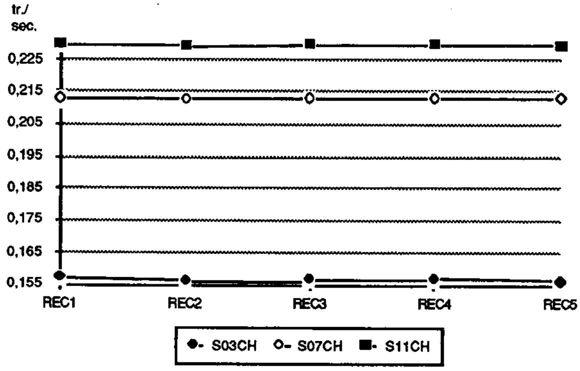


Fig. 6.11: System throughput by different checkpoint-oriented scenarios

While the system achieves almost the same throughput by the simulation of REC1, REC3, and REC4 in all crash-recovery scenarios, its throughput considerably decreases by the simulation of REC2 and REC5. Note that this decrease in the system throughput was reduced when REC2 and REC5 were simulated with a smaller log file (i.e. a 400 pages long ring file instead of a 800 pages long one). By realizing a smaller ring file on disk, REC2 and REC5 could reduce the number of objects and tuple sets stored on the log at any point in time. Therefore, less data had to be remapped onto database pages in case of a system crash. In this way, REC2 and REC5 could restore the state of the database faster, when the system crashed. We believe that REC2 and REC5 would allow even better system throughput, if log size had been more carefully chosen. The reduction of the log length has also affected transaction response time in S03CR. Figure 6.14 shows how the response time of the various transaction classes decreased when the log length was reduced by the simulation of REC2 and REC5. While the execution time of batch transactions and CHECKOUT operations was reduced by about 9.5%, the response time of CHECKIN operations was decreased in about 30 seconds (i.e. 27% of the total execution time).

Let us now analyze how the recovery mechanisms affect transaction response time in the different simulation scenarios. In most of the simulation runs, those recovery mechanisms which were realized at the server's page or tuple level showed a similar behavior. They affected the response time of the different transaction types almost in the same way. This behavior can in part be explained by the fact that these mechanisms require mapping operations to be processed as part of the transaction execution. Since mapping operations take very long, the differences among REC1, REC3, and REC4 affect transaction response time only marginally. The mechanisms which support deferred mapping, on the other hand, have influenced response time very much. Different transaction types were influenced differently, though. REC2 usually reduces the response time of update batch transactions by the cost of increasing the response time of other transactions (especially CHECKOUT operations). REC5, on the other hand, considerably reduces the response time of CHECKIN operations but increases the time of both read batch transactions and CHECKOUT operations. By allowing update transactions to be committed before their associated mapping operations take place, REC2 and REC5 force the response time of those transactions to be reduced. The deferred mapping operations, then, are executed in parallel to other transactions by demon processes. In this way, the server's multiprogramming level is increased, although the server's processing capacity remains the same. Consequently, the thrashing in the system increases and the execution of other transactions takes longer. The overload caused on the server by demon processes can be reduced only if the server's processing capacity is increased.

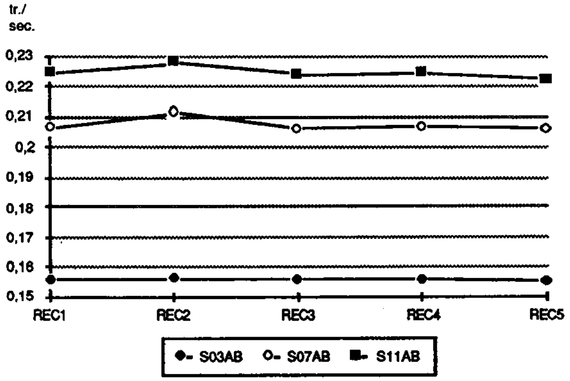


Fig. 6.12: System throughput by different transaction-abort scenarios

Figure 6.15 compares response times obtained by the simulation of the various recovery mechanisms in S03CH. This scenario presents both high design transaction and update transaction rates and long-duration processing activity at the workstation. Simulation results are compared on the basis of the transaction response times obtained with REC1. While REC2 reduces the processing time of update batch transactions by 25.2%, it increases the response time of CHECKOUT operations by 14%. Note that in scenario S03CH CHECKOUT operations represent 36.4% of the transaction load while update batch transactions make only 21.3% of the load. REC3 and REC4 produce similar response times. These times are comparable to the ones obtained by the simulation of REC1. Although REC3 logs less data than REC1, the former executes a more complex algorithm to manage its log buffer. The time REC4 saves by not having to generate checkpoints is offset by the extra I/O-operations executed during transaction commit. Although REC5 reduces CHECKIN response time by 54% and this operation contributes with 36.4% of the total transaction load, this recovery mechanism increases the response time of all other transaction types.

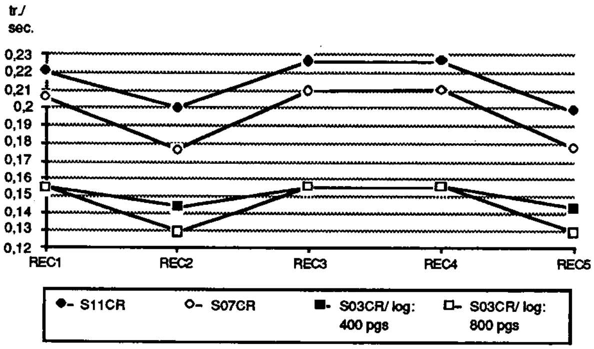


Fig. 6.13: System throughput by different crash-oriented scenarios

Let us now analyze transaction response time in environments which present low design transaction rates. Figure 6.16 shows response time differences which were observed by the simulation of S07CH. Besides presenting a low design transaction rate, this scenario shows a high update transaction rate and long duration processing activity at the workstation. Once again, the results obtained by the simulation of REC1 do not differ very much from those observed by the simulation of REC3 and REC4. Since most of the mapping operations in S07CH process sets of tuple instead of complex objects, they usually take shorter than, for instance, mapping operations in S03CH. Consequently, some differences among REC1, REC3, and REC4 can be better noted. REC3, for example, has better times for update batch transactions (which represent 53.1% of the load) and CHECKIN operations, because it logs less data than REC1. REC4 produced worse response times than REC1 for almost all transaction types, because it forces updates to disk during transaction commit. While the log file is written sequentially, REC4 uses random access to force udates to the database on disk. Sequential write operations on disk usually take shorter than random access, since in most cases the write head needs not to be moved (i.e. is kept over the same disk track). REC2 reduced the response time of most transactions by 14.1%. On the other hand, it increased the processing time of 38.8% of the transactions by almost 30%. REC5 reduced response time of CHECKIN operations by 60% but increased processing time of read batch transactions by 44%.

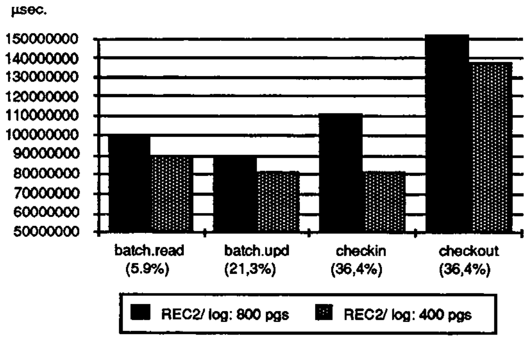


Fig. 6.14: REC2's crash-recovery performance by different log file sizes

Figure 6.17 depicts response time difference in S11CH. In this scenario, PROCESS R-Tr takes only 1 minute at the workstation. Therefore, objects are checked back into the public database sooner. Since S11CH presents a high transaction update rate, most of the objects checked in at the server node had first been updated by the designer at the workstation. Consequently, REC3 reduced the response time obtained by REC1 for CHECKIN operations, because it logs only updated tuples instead of whole pages (i.e. it executes less I/O-operations than REC1). The same occurred with the response time of update batch transactions. REC2 reduced CHECKIN response time in the same proportion it increased CHECKOUT response time. On the other hand, REC2 reduced the time of update batch transactions by 11.1% while increasing the processing time of read batch transactions by 42.8%. Finally, REC5 repeated its usual performance by reducing CHECKIN response time very much and increasing the response time of other transaction types.

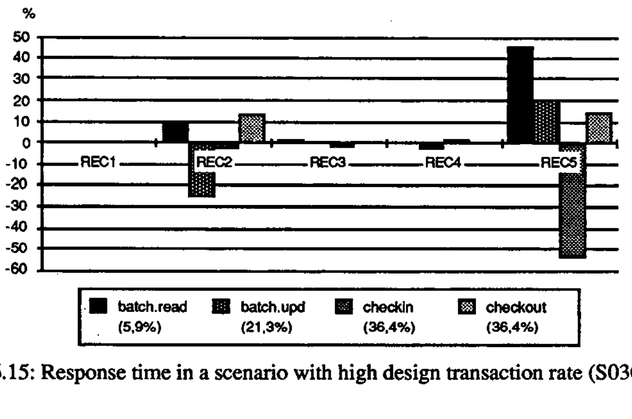


Fig. 6.15: Response time in a scenario with high design transaction rate (S03CH)

In scenarios which present both low design transaction and low update transaction rates, REC5 had the best performance. Figure 6.18, for instance, shows response time differences observed by the simulation of S09CH. Although REC5 increased the processing time of read batch transactions by 9.7%, this percentage represents a time increase of only three seconds in 30 seconds. On the other hand, REC5 reduced the response time of all other transaction types in S09CH. REC5's behavior can be explained by the fact that only a few transactions updated data in S09CH (i.e. only 36% of the executed transactions). Therefore, fewer demon processes were started in this scenario to execute mapping operations in parallel to other transactions. Consequently, the server was not so much overloaded.

Descending the scenario classification tree further, we now comment on how chained-I/O facilities affected both recovery and overall system performance. In most of the simulated scenarios, chained-I/O influenced recovery performance only marginally. As we already explained, the processing activity at the server node is CPU-bounded and not I/O-bounded. Waiting times and mapping operations dominated most of the transaction's execution time. It is possible, though, that chained-I/O operations could have exerted a greater influence on the system's performance, if we had simulated larger data objects in the public database so that the difference between the number of chained and normal I/O-operations would have increased. The simulated object clusters consisted of four database pages and each subobject read was stored onto two of these pages. Therefore, when chained-I/O was simulated objects were read into the buffer or written to disk in one I/O-operation. For the same read/write operation, two I/O-operations were needed when normal input/output devices were simulated. That is, chained-I/O reduced the number of access operations to disk by 50%. Nevertheless, the total number of this operations was always kept relatively low. On the other hand, chained I/O-operations always read two extra database pages into the page/segment-oriented buffer, since they always bring a whole cluster into main memory (although the objects being read are stored onto only two of the four cluster pages).

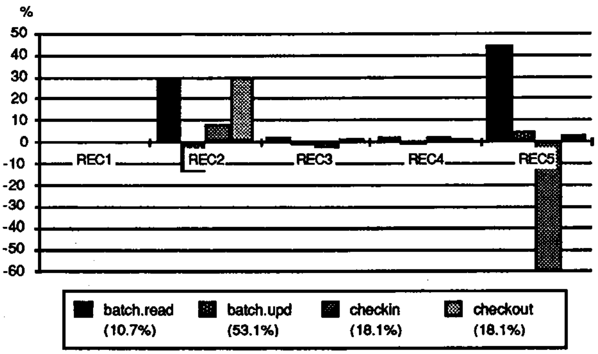


Fig. 6.16: Response time in a scenario with low design transaction rate (S07CH)

The curves for transaction response time in Figure 6.19 show that the difference in the number of I/O-operations had no much effect on recovery performance in checkpoint-oriented scenarios. The curves in Figure 6.19 respectively depict response time in S07CH and S08CH by the simulation of REC1 which realizes a page-oriented logging algorithm. Both scenarios present the same design transaction and update transaction rates, are checkpoint-oriented, and model long-duration design processing at the workstation. Only S07CH models chained I/O-operations, though. Chained-I/O have reduced the response time of read batch transactions a little. Since the processing time for mapping operations in these transactions was relatively short, the difference on the number of executed I/O-operations could be better noted. On the other hand, chained-I/O performed even worse than normal I/O-operations by the simulation of CHECKIN operations. This result is related to the fact that chained I/O-operations always brought 50% more data into the buffer than it was necessary. Therefore, the buffer manager was forced to flush more data to disk. This, in turn, increased the number of I/O-operations again. We believe that chained-I/O performance is closely related to the organization of data objects in clusters. To prevent that chained I/O-operations read too much unnecessary data into main memory, the physical organization of the database on disk must be carefully planned.

Figure 6.20 shows the effects of chained-I/O on recovery performance in a crash-oriented simulation scenario. By the simulation of crash-recovery activities, chained-I/O has clearly helped to increase recovery performance. The difference between chained-I/O performance in crash-oriented scenarios and in other scenarios can be explained by the fact that recovery mechanisms executed more I/O-operations by crash-recovery than by other recovery activities. It must be noted, though, that the increase of chained-I/O performance in crash-oriented scenarios showed not the same intensity by the simulation of recovery mechanisms which maintain smaller log files on disk as, for instance, REC3. Furthermore, chained-I/O became even less important by the simulation of recovery algorithms for which crash-recovery activities are CPU-bounded (e.g. REC2). Once again, we want to emphazise that the performance of chained-I/O would probably increase further, if we had simulated much larger data objects.

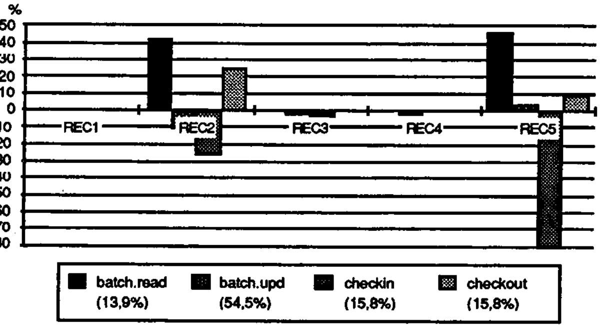


Fig. 6.17: Response time in a scenario with short processing times at the workstation

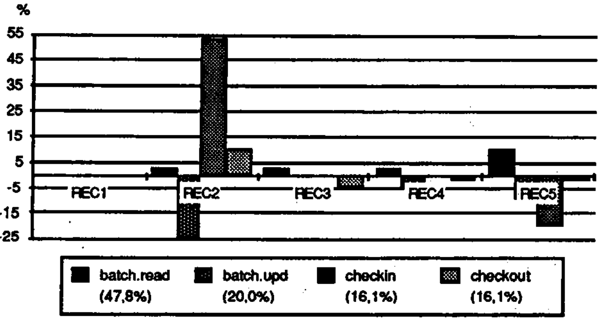


Fig. 6.18: Response time in a scenario with low update transaction rate (S09CH)

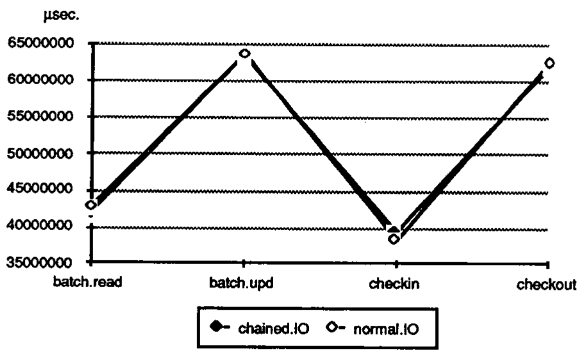


Fig. 6.19: Effects of chained-I/O in checkpoint-oriented scenarios (S07/S08CH)

6.4.5 Comparing Disk Space Occupancy for the various Recovery Mechanisms

As expected, the simulation results have shown that the page-oriented recovery mechanism (REC1) is the one which requires most disk space to store recovery information. Besides confirming this empirical expectation, though, the simulation study has also quantified the stable storage space consumed by each one of the simulated techniques. Figure 6.21 compares maximum and average log lengths for the five recovery mechanisms. These log sizes were observed by the simulation of S03NR. This scenario models a normal system operation environment which presents high design transaction and update transaction rates. By the simulation of this type of scenario, the recovery algorithms do not generate checkpoints. Therefore, the log size can only grow during system operation. Although the graphic of Figure 6.21 is based on the simulation of S03NR, the lengths of the different log files maintained this same proportion in almost all other simulation runs.

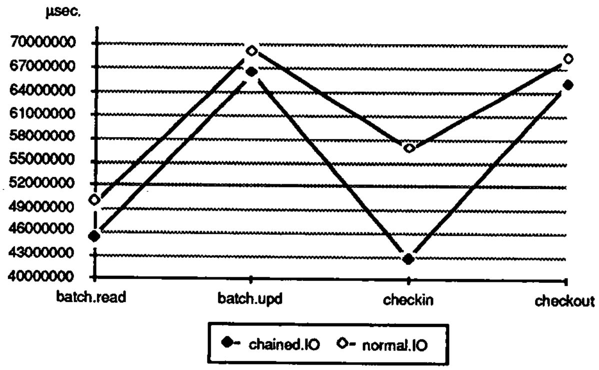


Fig. 6.20: Effects of chained-I/O in crash-oriented simulation (S07/S08CR)

By the simulation of REC1, the log size achieved the mark of 18489 logged data pages. The maximum size of the log file by the simulation of both REC2 and REC5 was 8150 pages. REC3's maximum log size was kept by 7576 pages. Since REC4 simply forces transaction updates to disk at commit time, it does not need to maintain extra recovery information on a log file. During commit, REC4 first writes the transaction updates to a special area on disk. This so-called intention list of updates is used in the case of a system crash to guarantee atomicity for the transaction-oriented checkpoint generated by REC4 on the basis of the FORCE strategy. The intention list on disk is discarded by REC4 as soon as the transaction commits. Therefore, intention lists were not considered to be a log file for the purposes of quantifying storage space consumption for the recovery mechanisms.

The simulation results show that, on the average, REC2 and REC5 consumed only 44% of the disk space used by REC1 to store recovery information. REC3's average space requirement on disk was even more modest (i.e. 41%). It must be noted, though, that while REC2 and REC5 saved complete data objects on the log, REC3 stored only updated records on that file. REC2 and REC5 could keep their log files small because they saved whole data objects or tuple sets as single log records. Thus, these algorithms did not write so much log control data to disk as REC3 did. The latter recovery mechanism stored every updated tuple on the log as a single record. Moreover, REC2 and REC5 could have kept the log even smaller, if only the updated parts of objects and tuple sets were to be saved on stable storage.

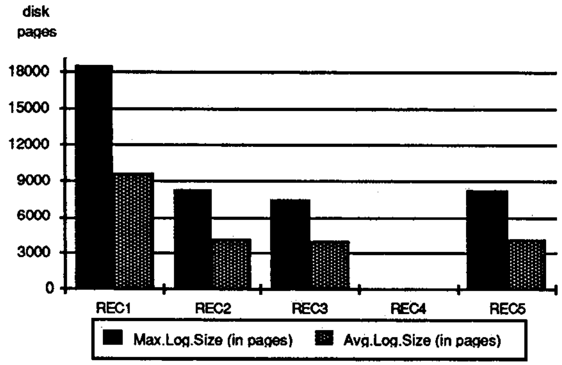


Fig. 6.21: Average and maximum disk space consumptions by normal system operation

6.4.6 Extra Simulation Scenarios with Low CPU Capacity

Based on the results obtained by the simulation of the 12 original scenarios (which are described in Figure 6.7), we decided to investigate recovery performance in the design environment under some other conditions. Besides analyzing the effects of increased server processing capacity on recovery performance, we were also interested in evaluating recovery in environments where only design transactions are processed. Moreover, scenarios where transactions present higher access locality were simulated, too. Finally, recovery performance was investigated in database systems which present lower costs for mapping operations related with design transactions. In the following, we report on some of the results obtained by the simulation of extra scenarios.

Figure 6.22 compares recovery performance on the basis of transaction response time when the public system processes only CHECKOUT and CHECKIN operations. REC3 and REC4 help response time of update transactions to decrease a little bit in comparison to the response times allowed by REC1 (i.e. respectively 0.8% and 0.5%). This can be explained for REC3 by the fact that this mechanism logs less data than REC1. Although REC4 also writes database pages to disk, it needs not to generate checkpoints as REC1 does. The reduction on response time for CHECKIN operations allows the designers at workstations to start new CHECKOUT operations earlier. By low CPU capacity on server, though, these operations are forced to wait longer in the server's ready queue. Consequently, the response time of CHECKOUT operations is a little bit higher by the simulation of REC3 and REC4 than it is by the simulation of REC1 (i.e. respectively 0.6% and 0.1%). The phenomenon described above occurs with much higher intensity by the simulation of REC5. Besides allowing CHECKIN operations to be committed before their respective mapping operations are executed by the public system, this mechanism let new remote operations be taken from the ready queue even before those mapping operations terminate. As a consequence of that, the number of short transactions being processed in parallel by the server strongly increases. New CHECKOUT operations, then, wait not in the ready queue but in the CPU queue much longer. The result of REC5's strategy in environments with low CPU capacity can be clearly seen in the graphic of Figure 6.22. While the processing time of CHECKIN operations is reduced by 71% percent (in comparison to REC1), the response time of CHECKOUT operations is increased by 39%.

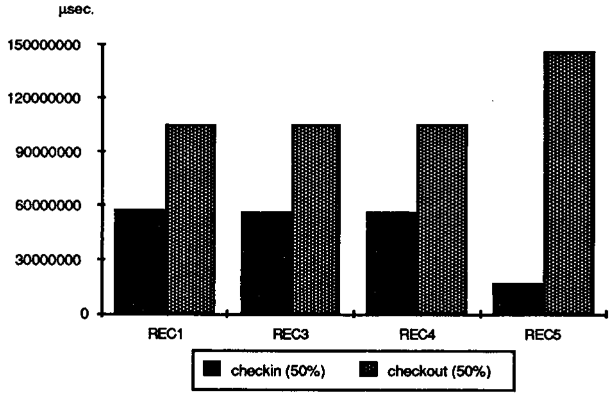


Fig. 6.22: Response time in a scenario where only design transactions are processed

By the simulation of a scenario where transactions present a higher locality of access (i.e. 30% instead of the 15% with which the other scenarios where simulated), we wanted to investigate if higher locality affects REC4's performance in the same way it does in business-oriented database systems. Furthermore, we also wanted to analyze the performance of recovery mechanisms which support deferred mapping under higher locality. Figure 6.23 relies on different transaction response times to compare recovery performance under 30% locality in a scenario which presents high design transaction and high update transaction rates (i.e. SE02CH). Since SE02CH presents low server processing capacity (i.e. 9 MIPS), both queue times and mapping operations dominated transaction processing time.for all transaction types. Consequently, the fact that REC4 forces the same database pages to disk more often in SE02CH than in other simulation scenarios had no much influence on the response time of the transactions processed. This can be concluded on the basis of the results showed in Figure 6.23. REC4 increased the response time for read batch transactions and CHECKIN operations only marginally in comparison to REC1. On the other hand, REC4 even reduced the processing time of update batch transactions by 3.9%. The behavior of all recovery mechanisms simulated in SE02CH did not differ very much from the behavior they had by the simulation of other scenarios. Maybe, REC2 was the only mechanism which somewhat benefited from the increased locality of access. By allowing designers to start CHECKOUT operations earlier, REC2 increased the chance which following transactions had of finding the data they needed in the buffer. On the other hand, REC2 reduced the number of CHECKOUT operations being aborted by the server as well as the number of blocked transactions at that processing node. Figure 6.24 shows that while the number of aborted CHECKOUT operations was kept by 1500 during the simulation of REC1, REC3, and REC4, this number dropped to about 1200 by the simulation of REC2. REC5 produced the highest number of aborted and blocked transactions, though. The combination of higher multiprogramming level (which is forced by REC5) with higher access locality naturally leads to more access conflicts among concurrent transactions.

Now, let us comment on the way shorter-duration mapping operations affect recovery performance in a simulation scenario which presents high design transaction rate and high update transaction rate. By the simulation of SE3CH, we reduced the cost of mapping operations for CHECKOUT and CHECKIN from 384000 to 128000 machine instructions per database record. To maintain 85% of the CPU capacity allocated to transaction processing activities, though, we increased the number of terminals in the system (i.e. this number was brought to 87). These changes affected the performance of recovery mechanisms based on defferred mapping in two ways. First, shorter mapping operations reduced the advantage of committing update transactions before mapping operations take place. Secondly, the combination of early commit and high number of workstations and terminals increased the time during which transactions were kept in the ready queue of the server node. Figure 6.25 compares transaction response time in SE3CH. While the very low CHECKIN response time which was achieved by REC5 when other scenarios were simulated is not shown by this figure, the great differences among response times for CHECKOUT operations which were also observed by the simulation of other scenarios were reduced during the simulation of SE3CH. That is, the performances shown by the different recovery mechanisms tend to converge when mapping operation times get shorter in processing environments which present high update transaction rates.

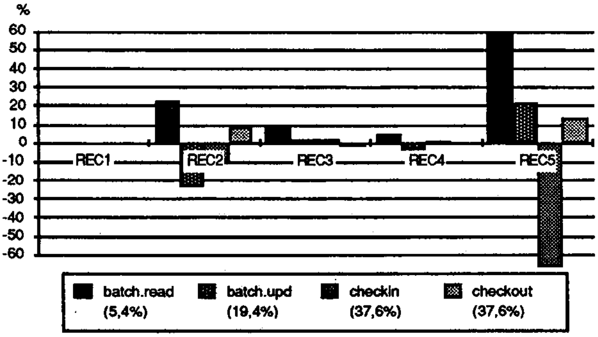


Fig. 6.23: Comparing response times when locality of access increased from 10% to 30%

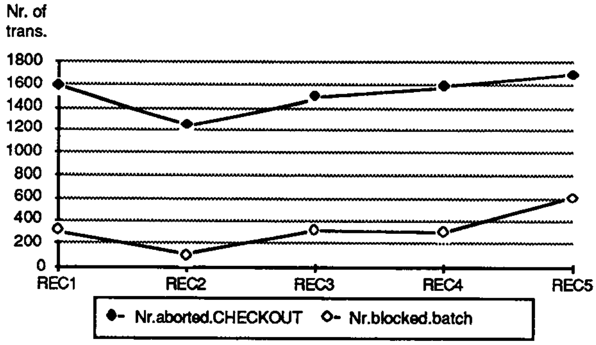


Fig. 6.24: Transaction failures in a scenario with high locality of access (30%)

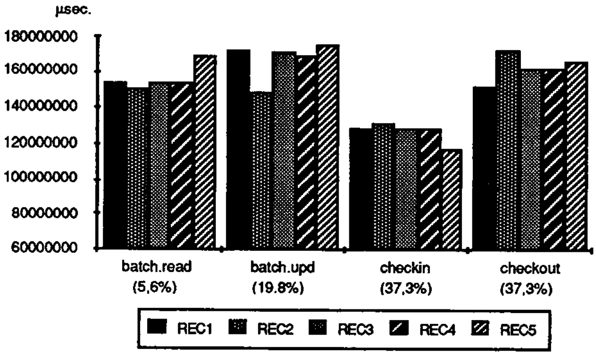


Fig. 6.25: Comparing response times when object mapping costs decrease

6.4.7 Simulating Higher CPU Capacity at the Server Processing Node

The simulation results presented above showed that recovery algorithms located at lower levels of the server node (e.g. REC1, REC3, and REC4) perform similarly in an integrated information system when the server's CPU capacity is relatively low (e.g. 9 MIPS). Since the system becomes overloaded by the execution of long-duration mapping operations, different recovery strategies such as FORCE and ¬FORCE or page logging and tuple/record logging exert no strong influence either on the overall system throughput or on the response time of the different transaction types simulated. On the other hand, simulation results also made clear that recovery algorithms based on deferred mapping cannot reduce the response time of all transaction types being simulated when the server's processing capacity is kept low. By the simulation of REC2, the processing time of update batch transactions decreased considerably but the waiting time for CHECKOUT operations in the server's ready queue increased significantly. By the simulation of REC5, the waiting time in the ready queue decreased but transactions got stuck waiting for the CPU.

Relying on the evaluation results obtained by the simulation of scenarios which present low server processing capacity, we decided to investigate the behavior of recovery algorithms when this capacity is increased. In the following, we comment on the results obtained by the simulation of S03CH with four different CPU capacities for the server: 9 MIPS, 11 MIPS, 25 MIPS, and 50 MIPS. As already explained, S03CH combines high design transaction rate with high update transaction rate, and long processing times at the workstation. Besides that, this scenario models the execution of chained I/O-operations. While no significant increase on throughput could be observed by the simulation of a 11 MIPS CPU, system performance was successively improved by the simulation of both a 25 and a 50 MIPS CPU at the server node. Figure 6.26 compares recovery performance on the basis of system throughput by high CPU capacities. By 25 MIPS, REC2 allows highest throughput. It is followed by REC5 and REC1. The high server capacity helped to reduce waiting times at the server's ready queue as well as at the CPU. The time saved by committing update transactions earlier was not completely transferred to read-only transactions anymore. By greater CPU capacity, the demon processes which executed mapping operations on behalf of committed update transactions did not interfere so much with other (concurrent) user transactions. By the simulation of a 50 MIPS CPU at the server node, REC5 shows the best performance and is followed by REC1 and REC2. By not allowing new transactions to be started before mapping operations for committed transactions terminate, REC2 prevents the public system from making use of the whole server capacity. On the other hand, REC3's lower performance can be explained by the fact that it has a more complex logging algorithm than REC1. Besides that, by the simulation of REC3 the number of aborted CHECKOUT operations increased. REC4 performs better by 50 MIPS than by 25 MIPS because the higher CPU capacity compensates REC4's longer I/O-operations.

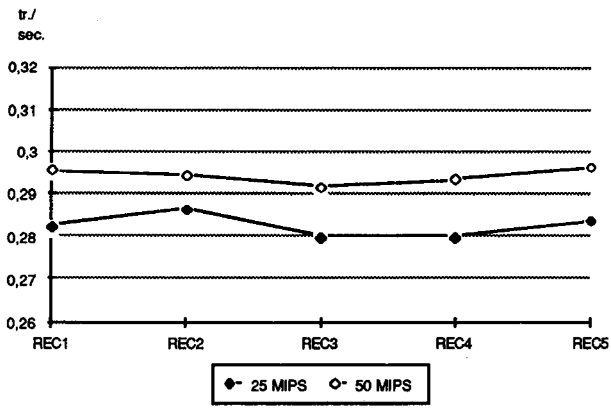
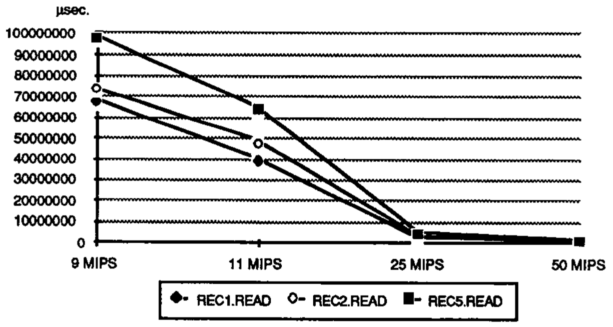


Fig. 6.26: System throughput by high CPU capacities at the server node

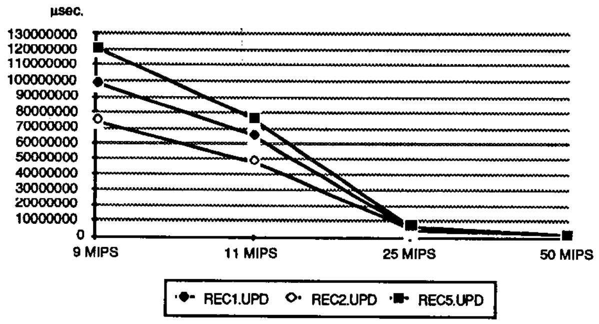
Recovery performance by increased CPU capacity can also be analyzed on the basis of transaction response time. Figure 6.27 shows how read batch transaction time varies when CPU capacity is increased at the server node. While the absolute time difference between response time under REC5 and REC1 decreases when CPU capacity is raised from 9 to 11 MIPS, the time difference between response time under REC2 and REC1 increases. On the one hand, REC5 helps the system to fully exploit the increase in CPU capacity by starting new transactions at the server in parallel to mapping operations for committed transactions. On the other hand, the server capacity is not high enough to reduce the extra waiting time in the ready queue that is induced by REC2. Consequently, REC2 cannot profit from the higher CPU capacity as much as REC1 does.



Note: read batch transactions represented only 5.9% of the transaction load simulated

Fig. 6.27: Response time for read batch transactions by different CPU capacities

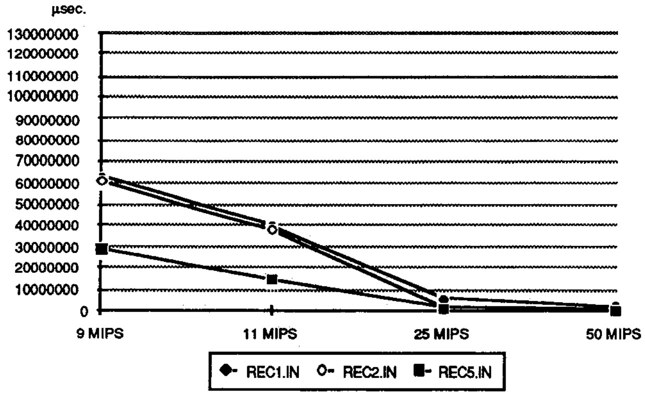
By 25 MIPS, all three recovery algorithms allow almost the same response time for read-only batch transactions. CPU capacity has become so high that the differences among the recovery mechanisms cannot be noted anymore. By this CPU capacity, response time for read batch transactions achieves the level of response times in business-oriented database systems (i.e. from two to five seconds). When CPU capacity is raised to 50 MIPS, transaction response time drops to about one second.



Note: update batch transactions represent 21.3% of the transaction load simulated

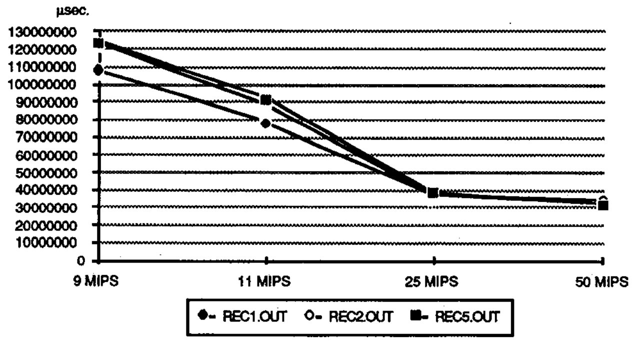
Fig. 6.28: Response time for update batch transactions by different CPU capacities

Figure 6.28 compares recovery performance on the basis of update batch transactions when the server´s processing capacity is increased. While the time difference between transaction response time under REC5 and REC1 decreases rapidly (e.g. from 21 seconds by 9 MIPS down to 10 seconds by 11 MIPS), the difference between REC1 and REC2 is reduced slowly (e.g. only five seconds by 11 MIPS). By increased CPU capacity, the complex algorithms of REC2 and REC5 are executed faster. Therefore, update transactions can be committed even earlier. Although REC1 also benefits from more powerful CPUs, it cannot commit update transactions so fast as REC2 and REC5 do. Similar to the process time of read-only transactions, response times for update batch transactions converge to almost the same value by 25 MIPS independently from the recovery algorithm being simulated.



Note: CHECKIN operations represented 36.4% of the server's load

Fig. 6.29: CHECKIN response time by different CPU capacities



Note: CHECKOUT operations represented 36.4% of the server's load

Fig. 6.30: CHECKOUT response time by different CPU capacities

Although CHECKIN response time decreases faster under REC1 and REC2 than under REC5, CHECKIN operations are more than 24 seconds shorter under REC5 by 11 MIPS anyhow (see Figure 6.29 for a comparison). Even when CPU capacity is raised to 25 MIPS, REC1 still performs worse than REC5. Only by 50 MIPS, all three recovery mechanisms allow (almost) the same response time for CHECKIN operations at the server node.

We now analyze how increased CPU capacity affects CHECKOUT response time in the various recovery environments simulated. The curves in Figure 6.30 show how response time for CHECKOUT operations decreases when the server's processing capacity increases. By 9 MIPS, response time under REC5 and REC2 is about 15 seconds longer than under REC1. By 11 MIPS, this difference drops down to 10 and 13 seconds for REC2 and REC5, respectively. On the basis of interpolation, we calculate that the difference in CHECKOUT response time by 18 MIPS would be around 6 seconds. By this CPU capacity, CHECKOUT operations under REC1 would take about 57 seconds at the server node. From the evaluations above, we can conclude that the advantage of REC5 concerning CHECKIN response time is kept further when CPU capacity increases, while the advantage of REC1 concerning CHECKOUT response time decreases in importance more rapidly. Although REC2 reduces the response time of update batch transactions even when CPU capacity is increased, this algorithm does not permit that the server fully utilizes the extra CPU power. Only REC5 really helps to improve parallelism at the server node.

6.4.8 Recovery Performance in a Design Cooperation Environment

As explained at the beginning of the present chapter, we also investigated recovery performance in a system which supports design cooperation on the basis of the GM3 processing model. Pursuing the goal of analyzing to which extent recovery can benefit from the buffer hierarchy realized by the design database system, we decided to test how deferred mapping can affect system performance in database systems which allow designers to exchange non-committed results via the group database located at the server node. For this purpose, we modified the simulation network so that every object being checked back into the group/public database was associated with a message sent from the workstation. This message was interpreted by **TM2** at the server node as the object arrived there. The message carried one of two contents. It either informed **TM2** that its associated object was completely processed by the designer at the workstation or told that simulation node that the object being checked in would soon be checked out of the group database by another designer to be processed further. The transaction manager interpreted the received message and informed the other simulation nodes of the server about its meaning. We simulated this version of GM3 with two different CPU capacities for the server node: 9 MIPS and 11MIPS. In all simulation runs, 20% of the objects being checked into the group database were associated with messages which informed **TM2** that those objects would soon be checked out again.

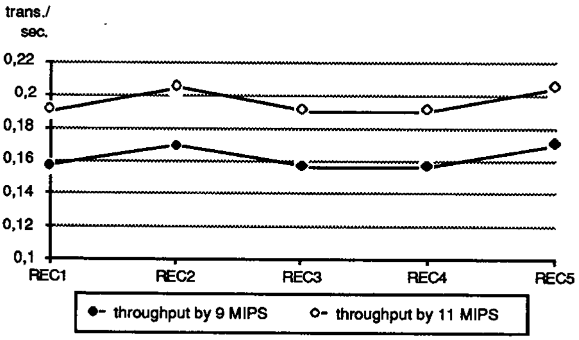


Fig. 6.31: System throughput related to CPU capacity in a design cooperation scenario

The graphic in Figure 6.31 compares system throughput in the GM3 scenarios simulated. The curves in the graphic show that REC2 and REC5 allowed best throughputs in both the 9 MIPS and the 11 MIPS environments. These two recovery mechanisms were the only ones which could understand the different semantics that CHECKIN operations could have. By processing a CHECKIN operation, REC2 and REC5 first saved the updated object on the log and then analyzed the message associated with it. If the object was to be checked out again soon, these recovery mechanisms sent it back to **TM2** instead of directly sending it to the MAP simulation node. By the next time the object was checked out, it was read from the log. The other recovery algorithms could not reduce the number of mapping operations on the basis of the messages received from the workstation. Since REC1, REC3, and REC4 are realized at lower system levels, they must force mapping operations to take place in order to save data updates by CHECKIN.

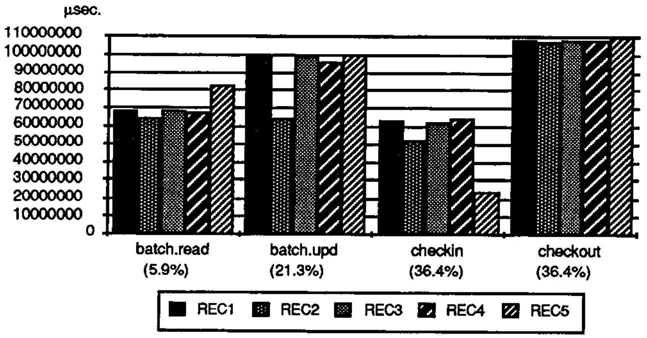


Fig. 6.32: Response time in a design cooperation environment with low CPU capacity

The better performance of REC2 and REC5 in design cooperation environments can also be observed by the transaction response times these mechanisms allowed there. Figure 6.32 depicts response times in the 9 MIPS GM3 scenario. For all transaction types, REC2 allowed the lowest response time. Although REC5's performance related to both CHECKOUT and update batch transaction response times improved, it could not guarantee a better response time for read batch transactions. By 9 MIPS, REC5 still causes the server to become overloaded, even if the number of design mapping operations is reduced in the system. On the other hand, REC5 reduced CHECKIN response time even further.

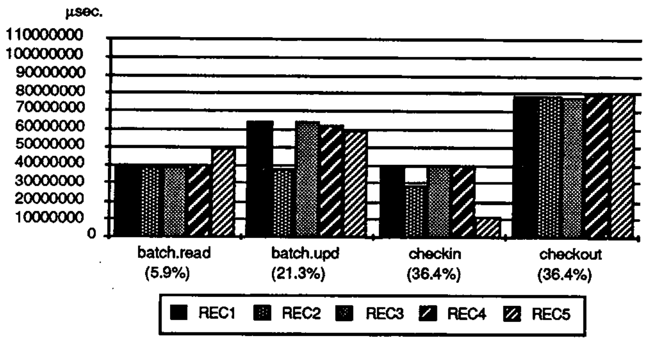


Fig. 6.33: Response time in a design cooperation environment with higher CPU capacity

The graphic in Figure 6.33 compares transaction response time for the 11 MIPS GM3 scenario. Although REC2 still showed the best overall performance, the differences among response time for both CHECKOUT and read batch transactions were reduced. The former became even shorter under REC1 and REC3. As already observed in subsection 6.4.7, all other recovery mechanisms can better benefit from small increases in the CPU capacity than REC2 does. While the performance bottleneck of other recovery mechanisms is represented by high waiting times in the CPU queue and other internal queues of the server node, REC2's performance depends on the reduction of the waiting time in the server's ready queue. This waiting time can only be reduced, though, if CPU capacity is raised further. The performance of REC5, on the other hand, improved a lot as CPU capacity reached 11 MIPS. Response time for read batch transactions decreased about 34 seconds (i.e. from 83 down to 49 seconds), while update batch transaction time became lower under REC5 than under REC1, REC3, or REC4.

6.4.9 Summarizing the Main Results of the Simulation Study

In this subsection, we recapitulate the main results of the recovery performance evaluation presented in the prior subsections. Moreover, we relate these results to the empirical recovery performance study made in chapter 5 and to the goals of the simulation study which were stated at the beginning of the present chapter.

Through the simulation of five different recovery techniques in an integrated information system, we further investigated four of the suppositions made at the end of chapter 5:

- Opposed to recovery in business-oriented database systems, recovery algorithms based on FORCE and those algorithms based on $\neg$FORCE may perform similarly in design database systems.
- Page-oriented logging mechanisms will probably perform as well as record-oriented logging mechanisms at the server node of design database systems.
- Object-oriented logging mechanisms may perform well in design database systems which realize buffer hierarchies.
- Object-oriented logging mechanisms which support deferred mapping operations at the server node will probably perform well in design cooperation environments.

Besides these four suppositions, we were interested in investigating the following more general questions: how recovery mechanisms perform in integrated information systems which support both business-oriented as well as design transactions; at which level of abstraction the recovery mechanism should be implemented at the server node to guarantee best recovery performance; how recovery algorithms can help to improve overall system performance in integrated information systems. Both the suppositions and the questions above were investigated in great detail through the recovery simulation study described in the prior sections. The main results which were obtained by this study are listed below.

- The burden represented by the recovery activity in the system is not very significant when compared with the cost of other system activities (e.g. buffer management, data representation mapping operations). Therefore, system performance could not be improved very much by simply reducing costs associated with recovery activities (e.g. number of I/O-operations). Consequently, most of the simulated recovery mechanisms performed similarly. Only those recovery mechanisms which support deferred mapping operations significantly affected system performance.
- The response time of business-oriented transactions increase very much in integrated information systems which also support design transaction execution. The time which short transactions spend in both the server's ready queue and CPU queue is proportional to the number as well as to the duration of the design operations being processed in parallel to the short transaction at the server node. By lower CPU capacity on server (e.g. 9 MIPS), up to 43% of the response time of batch transactions was waisted at the server's ready queue.
- Without increasing the server's capacity, the only recovery algorithms which affected waiting time in the ready queue and in the CPU were REC2 and REC5. The former increased waiting time in the ready queue by allowing new transactions and remote

design operations to be started earlier at terminals and workstations but preventing new transactions to be taken from the ready queue until mapping operations related to committed update transactions terminated. REC5 reduced waiting times in the ready queue by permitting new transactions to execute in parallel to mapping operations for committed transactions. By doing that, REC5 increased the waiting times in the CPU queue of the server node, though.

- Since mapping operations dominated processing time at the server node, the number of I/O-operations started by each of the recovery mechanisms being tested did not affect system performance significantly. Consequently, recovery mechanisms based on FORCE (e.g. REC4) performed similar to those based on ¬FORCE (e.g. REC1). Moreover, transaction response time as well as system throughput were affected by page-logging only in especial situations (e.g. when processing time at the workstation was kept short and design transaction rate was kept low in the load).
- In transaction-abort-oriented scenarios, system throughput was usually increased by the simulation of REC2. In crash-oriented scenarios, on the other hand, both REC2 and REC5 reduced system throughput. This decrease in system throughput could be in part avoided when the log size for REC2 and REC5 was reduced on disk. In checkpoint-oriented simulation scenarios, all recovery mechanisms presented similar performance.
- Especially by low CPU capacity on server, no one of the recovery algorithms simulated could present best response times for all transaction types at the same time. As already explained, REC1, REC3, and REC4 showed very similar performance almost always. In scenarios with low update transaction rate, REC1 reduced response time for read batch transactions while REC4 reduced the time of update batch transactions and CHECKIN operations. In those scenarios, REC2 reduced CHECKOUT response time a little bit. In scenarios with low design transaction rate and short processing times at the workstation, REC3 performed better than REC1 and REC4, because the number of I/O-operations executed in those scenarios became important, since the number and the duration of mapping operations decreased.
- In almost all simulation scenarios, REC2 significantly reduced the response time of update batch transactions. On the other hand, this mechanism increased response time for both read-only batch transactions and CHECKOUT operations. In most of the simulation runs, REC5 reduced CHECKIN response time by more than 50% by the cost of increasing response time for CHECKOUT operations and batch transactions. These results show that recovery based on deferred mapping cannot uniformly increase overall system performance when CPU capacity is kept low at the server processing node.
- The simulation study has shown that the chained-I/O facility does not always help to improve system performance. If the data organization on disk is not carefully planned and maintained, chained I/O-operations may even reduce system performance by forcing the buffer manager to replace more database pages in the buffer than necessary. By the simulation of crash-oriented scenarios, though, chained-I/O has proven to be a good help. Especially by recovery mechanisms which maintain large log files (e.g. REC1), chained I/O-operations helped transaction response time to be kept shorter.
- Page-oriented logging algorithms require twice as much space on disk than record-oriented or object-oriented logging algorithms. Object-logging saves space on disk by writing object updates together to the log file as single log records. In this way, less log control information (e.g. log record type information) must be stored on the log.
- Opposed to its behavior in business-oriented database systems, recovery based on FORCE (i.e. REC4) did not show a worse performance when the access locality of transactions was increased from 10% to 30%. It still performed very similar to recovery based on ¬FORCE (e.g. REC1). This simulation result can be explained by the fact that I/O-operations did not affect system performance very much by the simulation of scenarios where a great number of long-duration mapping operations are processed.

- By a higher access locality, REC2 improved system performance. This algorithm allowed new transactions to be started earlier in a scenario where transactions have more chance of accessing the same data. Therefore, the probability with which data being accessed could already be found in the buffer increased.
- REC2 and REC5 increase system throughput more than the other recovery mechanisms simulated when the server´s processing capacity is increased.
- As the CPU capacity at the server increases, REC5 becomes more attractive, since it increases parallelism at that processing node. Its higher response times for read-only transactions decrease faster than its advantage concerning CHECKIN response time.
- Recovery mechanisms which support deferred mapping operations proved to perform much better in design cooperation environments. Since these mechanisms log high-level data abstractions, they can take advantage of the application semantics and identify when transaction updates must be immediately mapped onto database pages and when the related mapping operations can be delayed (and even not executed). By the simulation of design cooperation scenarios, REC2 uniformly improved response time for all transaction types. Besides that, both REC2 and REC5 allowed best system throughput.

Chapter 7

Conclusions

7.1 Summarizing the Dissertation

The present work reported on the investigation of recovery requirements in design database systems which realize hierarchies of system buffers. This investigation was based on the architecture of various existing design database system prototypes as well as on a set of well known design processing models. The research work pursued the following goals:

- The identification of main database recovery requirements posed by the design environment.
- The analysis of the behavior of existing database recovery techniques in the design environment.
- The investigation of possible characteristics of the design environment that either prevent existent recovery algorithms from working properly or force them to be modified in order to be integrated into design database systems.
- The evaluation of recovery performance in design database systems which realize hierarchies of system buffers.

The first chapter of the dissertation explained how database requirements posed by design applications differ from those of business-related applications. While the database of business-related applications can be modeled by simpler data models supporting only less structured, flat data records (e.g. tuples in the first normal form) and a few types of relationships among them, design applications need more powerful data models which allow the user to define and manipulate highly structured data objects (e.g. molecules). Moreover, in these applications the user can process data for long periods of time (e.g. days, weeks). In business-oriented applications, on the other hand, data processing activity usually relies on the (conventional) transaction paradigm. Opposed to data processing in design applications, conventional transactions typically execute in a few seconds. A third important difference between conventional and design database environments is that the latter ones may permit users to exchange results of non-committed transactions, while transactions in conventional database systems execute under strict isolation.

Chapter 2 reviewed the architecture of some existing non-standard database system prototypes and discussed the way they try to cope with novel database requirements posed by design applications. Most of the prototypes reviewed realize a multi-level database system architecture on the basis of a server-workstation computer configuration. By introducing the notions of public and private databases, these systems realize a database hierarchy which supports the isolation of parts of the database for long periods of time. The user copies the desired data objects from the public database on server into his private database at the workstation (issuing CHECKOUT operations). While the user updates the object copies at the workstation, the original object versions remain locked in the public database. At the end of his work, the user issues so-called CHECKIN operations at the server node to integrate updated object copies into the public database and to release the locks related to them.

Most of the prototypes reviewed in chapter 2 also realize what we decided to call buffer hierarchies. To accelerate data processing activities at the system's application level, these prototypes implement so-called object-oriented buffers in higher system layers. By storing data in a main memory representation known by the application, the object-oriented buffers help the database system at the workstation to avoid costly representation mapping operations between the application work space and the page/segment buffer at the storage system layer. At the end of chapter 2, we proposed a reference system architecture for design database systems which is based on the architectures of the prototypes reviewed.

The study of design database system characteristics which was initiated in the first two chapters of the dissertation, was complemented in chapter 3. There, we analyzed the properties of some well known design processing models which were proposed in the literature. Relying on some basic properties of each of those models, we identified three different classes of design processing models and generalized them by proposing three so-called general design models. The first one of them (GM1) models the user's design work at both server and workstation as a set of independent work steps, each one of them related to a specific data object. The user can check objects out of and into the public database at any time. GM2 models the user work at the server processing node as an atomic transaction. CHECKOUT and CHECKIN operations follow a strict two-phase protocol. All objects updated at the workstation are checked back into the public database atomically. The third general model proposed supports design cooperation environments. That is, designers belonging to the same group can exchange semi-committed results. Together with the reference architecture proposed in chapter 2, the general models presented in chapter 3 served as a basis for the investigations reported in the next chapters of the dissertation.

In chapter 4, a thorough investigation of possible failures in the design evironment was carried out. We discussed both expected and unexpected failures, suggested where in the overall design system each specific failure type should be dealt with, and proposed a failure model for design database systems. Relying on this failure model as well as on the reference architecture and the general design processing models, a set of recovery protocols was presented which should serve as a basis for the realization of database recovery mechanisms in the design environment. Each recovery protocol presented copes with a specific failure type or recovery situation in the design environment. Some recovery situations supported by the protocols are transaction backout, deadlock, and system crash at the server node as well as savepoint generation and design transaction backout at the workstation. Besides, we distinguished recovery protocols which both rely on and guarantee transaction serializability from those which are based on objectwise two-phase lock. The latter ones support design environments which allow designers to exchange semi-committed results (i.e. realize GM3). To control the execution and recovery of related design transactions in GM3, we proposed a set of algorithms which are based on a directed graph representing transaction relationships in the design environment. On the basis of the so-called group transaction graph, the system can manage (i.e. synchronize and recover) transactions in design cooperation environments.

In chapter 5, we analyzed the correctness of existing database recovery algorithms in the various subsystems of the design environment. Besides that, an empirical performance evaluation of various recovery algorithms was carried out on the basis of the recovery requirements derived in the previous chapters of the dissertation. We extended the classification of recovery algorithms proposed in [HäRe83] to better capture specific properties of recovery in the design environment. Therefore, besides analyzing recovery mechanisms on the basis of the propagation strategy they support, their behavior at transaction commit time, the kind of buffer management strategy they cope with, and the way they produce checkpoints, we also considered recovery properties related to both the particular architecture of design database systems and characteristics of design applications. We discussed, for instance, whether recovery mechanisms which integrate recovery activities at the server node and at the workstation can perform better than recovery algorithms which perform their activities at the server and at the workstation separately. On the other hand, we distinguished the recovery algorithms which can support design cooperation environments from those which are based on transaction serializability. Finally we classified the algorithms on the basis of the transaction paradigm they follow: the conventional transaction paradigm or the nested transaction concept.

Besides empirically analyzing recovery performance in the design environment, we also proposed some extensions to existing recovery techniques so that they can support either design cooperation environments or recovery in server-workstation networks where data objects are exchanged from one processing node to the other via CHECKOUT and CHECKIN operations.

Some of the conclusions (and expectations) to which we came by means of the empirical performance evaluation carried out in chapter 5 are listed below.

- It is expected that integrated recovery algorithms at the server and the workstation will perform worse than isolated recovery mechanisms for each processing node. We believe that the former algorithms can overload the communications subsystem and the server itself.
- Recovery mechanisms which support deferred mapping operations at the server node should perform well in design cooperation environments. Since these mechanisms are realized at higher system levels (e.g. object-oriented level), they can easily capture the different semantics of CHECKIN operations. These mechanisms can, for instance, distinguish CHECKIN operations which bring committed object versions into the public database from those operations which only integrate non-committed object versions into the group database. In some cases, the mapping operations related to a CHECKIN operations of the latter type can be, at least temporarily, avoided.
- Opposed to the performance they present in business-oriented database systems, recovery algorithms based on FORCE and those based on ¬FORCE should perform similarly at the server node of a design database system. The same phenomenon can also occur between page-oriented logging algorithms and algorithms which log higher data abstractions (e.g. data records). The expected reduction on performance differences caused by the number of I/O-operations produced by each recovery technique can be explained by the fact that mapping operations will probably dominate transaction processing time at the server node. Consequently, the server will become CPU-bound instead of I/O-bound. Therefore, the influence of I/O-operations on system performance tends to diminish.

The simulation study presented in chapter 6 relied on some of the results of the empirical performance evaluation made in chapter 5. We decided to simulate some specific recovery mechanisms in order to prove the results of that performance evaluation. For this purpose, we designed and implemented a simulation network which models an integrated information system supported by a design database system. This database system is distributed over a server-workstation computer system. The database system supports both business-oriented (batch) transactions as well as design transactions. In this processing environment, we executed a great number of simulation experiments to compare the performance of different recovery algorithms in the desing environment. We were mainly interested in comparing FORCE and ¬FORCE as well as page-oriented logging and record-oriented logging at the server node. Moreover, we also wanted to investigate the performance of object-oriented recovery mechanisms which support deferred mapping operations and, consequently, allow update transactions to commit at the server even before the (costly) mapping operations related to them are processed. We simulated recovery activities only at the server node of the system. On the other hand, we investigated recovery performance in GM1 as well as in GM3. Some of the results obtained by the simulation study are listed below.

- Recovery activities represent no significant burden to the system when CPU capacity is relative low (e.g. 10 MIPS). This simulation result can be explained by the fact that mapping operations are very expensive and represent the greatest burden in the design environment.
- Transaction response time can strongly increase in integrated information systems when very different transaction types are supported. Unless CPU capacity is relatively high (e.g. 25 MIPS in our simulation study), batch transactions must have to wait for design operations (e.g. CHECKOUT) to be processed on the server.
- In most of the simulation runs, the recovery algorithm which followed the FORCE strategy (REC4) achieved a performance which is comparable with the performance shown by algorithms which are based on ¬FORCE. Opposed to results obtained for conventional database environments, forcing updates to disk at transaction commit does not significantly reduce system throughput or increase response time in the design environment. Since mapping operations take much longer than I/O-operations, the operation of forcing results to disk at commit time represents only a small fraction of the overall transaction processing time.
- Recovery mechanisms which allow transactions to commit before their updates (in main memory representation) are mapped onto database pages reduce response time for all transaction types only in processing environments which present high CPU capacity. Otherwise (i.e. by low CPU power), these mechanisms reduce response time for updating transactions but increase the time of read-only transactions when compared with recovery mechanisms which save transaction updates only after the corresponding mapping operations have taken place. By low CPU capacity, much of the time saved for update transactions through deferred mapping operations is transferred to other transactions executing in parallel.
- Besides the recovery algorithm based on FORCE (REC4) which maintains no log file at all, algorithms which save data records (REC3) or complete updated objects (REC2, REC5) needed significantly less space in stable storage than algorithms which save updated pages (REC1).
- By low CPU capacity, the choice of a recovery mechanism for the server node of a design database system which supports an integrated information system will depend on the specific needs of the applications being supported and on the transaction load being processed. While recovery mechanisms based on deferred mapping can significantly reduce the response time of update transactions (e.g. REC2 reduces the processing time of update batch transactions in most cases and REC5 reduces CHECKIN response time), the more conventional recovery mechanisms (e.g. those which execute at the page level) usually guarantee better response times for read-only transactions.
- By higher CPU capacity (e.g. 11 or 15 MIPS in our simulation study), recovery mechanisms based on deferred mapping performed somewhat better than the other ones at the server node. When CPU achieved a certain level of extra capacity (25 MIPS in our simulation study), all recovery algorithms showed similar performance.
- In design cooperation environments, recovery algorithms based on deferred mapping always performed better than the other algorithms. The former could benefit from the semantics of CHECKIN operations to avoid the execution of unnecessary (and costly) mapping operations. On the basis of the simulation results, we believe that transaction management in design cooperation environments can be efficiently realized on the basis of the group transaction graph (G) presented in chapter 4 and of an object-oriented recovery mechanism based on deferred mapping.

7.2 Comparison with other Works

Although there exists a number of published works which report on recovery algorithms for design database systems, most of them analyzed recovery requirements and describe recovery mechanisms only for specific database systems (e.g. [KaWe84], [KLMP84], [WeKa84], [KoKB87], [GaKi88], [Ries89]). Other works treated recovery in design database systems in a more general way but analyzed recovery requirements only for simpler design processing models (e.g. [Kelt88]). The present work has investigated recovery requirements in the design environment on the basis of various general design processing models as well as relying on a system architecture derived from the study of various representative design database system prototypes. Furthermore, we investigated the behavior of both existing and newly proposed recovery techniques in the design environment.

Also a number of articles reporting on various recovery performance evaluations can be found in the literature. Most of them investigated recovery performance in centralized, business-oriented database systems, though (e.g. [Reut84], [AgD85b]). Recovery in multiprocessor database machines was investigated in [AgD85a]. In this study, only processing environments supporting short-duration transactions were considered, though. In [Wei87a] and [Wei87b], the performance of multi-level transaction management was compared with the performance of single-level transaction management. Although this research work compared recovery mechanisms in a multi-level system architecture, neither buffer hierarchies nor long-duration transactions were considered in the investigation.

We are not aware of other works which have evaluated recovery performance in an integrated database environment supporting conventional as well as long-duration, design transactions on the basis of a multi-level system architecture which realizes a hierarchy of buffers and is distributed over a server-workstation computer system. Moreover, this research work seems to be the first one which evaluates database recovery performance on the basis of three different criteria, namely, transaction response time, system throughput, and log size.

Although recovery mechanisms which save data at higher levels of abstraction (e.g. [Lind79], [ARI89a]) as well as multi-level recovery algorithms (e.g. [Verh79], [Wei89a]) have already been proposed in the literature, this work is the first one to analyze the behavior of recovery mechanisms which save recovery information before mapping operations take place. On the other hand, the deferred update techniques for database systems which have been proposed in the literature (e.g. [Camm81], [DLPS85]) do not consider the possible existence of buffer hierarchies.

The idea of controlling transaction cooperation on the basis of a directed graph has already been proposed in [PROF85]. Transaction management in [PROF85] differs from the approach presented here in, at least, two ways, though. First, concurrency control in PROFEMO is based on transaction serializability while our approach relies on object-oriented two-phase lock. Secondly, transactions waiting to commit in PROFEMO can either commit or abort. To better support cooperative design environments, the approach proposed here allows transactions in the ready state to be brought back into the active state so that the designer can decide what to do next.

7.3 Some Open Questions and Plans for Future Work

This work reports on a performance evaluation of recovery techniques in the design environment. We mainly investigated recovery performance only in environments which realize the GM1 processing model, though. Following investigations should also consider both GM2 and GM3 environments in more detail. Especially for the GM3 environment where transaction cooperation is allowed, the performance of recovery mechanisms relying on the group transaction graph presented in chapter 4 should be investigated more thoroughly.

The simulation results already showed that recovery mechanisms which are realized at higher levels of abstraction can benefit from the semantics of the application. Although we could quantify this benefit for some design cooperation scenarios, the investigation of how recovery mechanisms can both get and process information concerning the application behavior also constitutes a very important and interesting topic of research which was not covered by this work.

For all design processing models, recovery performance in environments with high CPU capacity should be analyzed in more details. Besides that, recovery performance in distributed server systems should be modeled and evaluated.

By simulating design environments which realize the GM2 processing model, recovery techniques for nested transactions should be investigated, too. The coordination of recovery actions at the server and at the workstation costitutes a very important area of study which has not deserved much attention until now.

Finally, this work has not investigated recovery performance at the workstation at all. The performance of new recovery techniques (as, for instance, the one in [Ries89]) should be compared with that of already existing recovery algorithms in this environment. Besides, new recovery techniques for workstations that consider the existence of other workstations in the system should be investigated. These techniques could alleviate transaction load at server node in systems where not all workstations have own disk units.

Bibliography

[AgCL87] Agrawal, R.; Carey, M.J.; Livny, M.: Concurrency Control Performance Modeling: Alternatives and Implications. ACM Trans. on Database Systems, Vol. 12, No. 4, December 1987

[AgD85a] Agrawal, R.; DeWitt, J.D.: Recovery Architectures for Multiprocessor Database Machines, Proc. ACM SIGMOD Int. Conf. on Management of Data, Austin, Texas, 1985

[AgD85b] Agrawal, R.; DeWitt, J.D.: Integrated Concurrency Control and Recovery Mechanisms: Design and Performance Evaluation. ACM Trans. on Database Systems, Vol. 10, No. 4, December 1985

[ARI89a] Mohan, C. et al.: ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. IBM Research Report RJ6649, IBM Almaden Research Center, January 1989

[ARI89b] Rothermel, K.; Mohan, C.: ARIES/NT: A Recovery Method Based on Write-Ahead Logging for Nested Transactions. IBM Research Report RJ6650, IBM Almaden Research Center, January 1989

[BaBu84] Batory, D.S.; Buchmann, A.P.: Molecular Objects, Abstract Data Types, and Data Models: A Framework. Proc.10th Int. Conf. on Very Large Data Bases, Singapore, 1984

[BaKK85] Bancilhon, F.; Kim, W.; Korth, H.F.: A Model of CAD Transactions. Proc. 11th Int. Conf. on Very Large Data Bases, Stockholm, 1985

[Banc88] Bancilhon, F.: Object-Oriented Database Systems. Proc.7th ACM SIGART-SIGMOD-SIGACT Symposium on Principles of Database Systems. Austin, Texas, March 1988

[BaRa88] Badrinath, B.R.; Ramamritham, K.: Synchronizing Transactions on Objects. IEEE Trans. on Computers, Vol. 37, No. 5, May 1988

[BeHG87] Bernstein, P.A.; Hadzilacos, V.; Goodman, N.: Concurrency Control and Recovery in Database Systems. Addison-Wesley Series in Computer Science, Addison-Wesley Publishing Company, 1987

[BrMa88] Brodie, M.L.; Manola, F.: Database Management: A Survey. In: Fundamentals of Knowledge Base Management Systems, J.W. Schmidt and C. Thanos (eds.), Springer Verlag, 1988

[Camm81] Cammarata, S.: Deferring Updates in a Relational Data Base System. Proc. 7th Int. Conf. on Very Large Data Bases, 1981

[Care83] Carey, M.J.: Modeling and evaluation of database concurrency control algorithms. Ph.D. dissertation, Computer Science Division (EECS), University of California, Berkeley, Sept. 1983

[CaSt84] Carey, M.J.; Stonebraker, M.: The performance of concurrency control algorithms for database management systems. Proc. 10th Int. Conf. on Very Large Data Bases, Singapore, 1984

[CODA71] CODASYL Database Task Group Report. Association of Computing Machinery, New York, 1971

[Codd70] Codd, E.F.: A Relational Model for Large Shared Data Banks. Comm. ACM, Vol.13, No. 6, June 1970

[DAM86a] Gotthard, W.: DAMOKLES - The Data Model of the UNIBASE Design Database System. Research Report 3/86, Forshungszentrum für Informatik, Karlsruhe, W.Germany, 1986 (in German)

[DAM86b] Raupp, T. et al.: The DAMOKLES System Architecture. Cooperation Project UNIBASE, Research Report, Forshungszentrum für Informatik, Karlsruhe, West Germany, 1986 (in German)

[DAM88a] Rehm, S. et al.: Support for Design Process in a Structurally Object-Oriented Database System. Proc. 2nd Int. Workshop on Object-Oriented Database Systems, Bad Münster am Stein-Eberburg, West Germany, September 1988

[DAM88b] Abramowicz, K. et al.: Software Distribution in Structured Object-Oriented Database Systems for Design Environments. Research Report, Forschungszentrum für Informatik, Karlsruhe, West Germany, 1988 (in German)

[Date83] Date, C.J.: An Introduction to Database Systems. Addison-Wesley Systems Programming Series, Addison-Wesley Publishing Company, 1983

[Davi78] Davies, C.T.: Data processing spheres of control. IBM Systems Journal, Vol. 17, No. 2, 1978

[DeOb87] Deppisch, U.; Obermeit, V.: Tight Database Cooperation in a Server-Workstation Environment. Proc. 7th Int. Conf. on Distributed Computer Systems, Berlin, 1987

[Depp86] Deppisch, U. et al.: Considering Database Cooperation between Server and Workstation. In: Informatik Fachberichte, Vol.126, Springer Verlag, 1986 (in German)

[DePS86] Deppisch, U.; Paul, H.-B.; Schek, H.-J.: A Storage System for Complex Objects. Proc. Int. Workshop on Object-Oriented DBS, Pacific Grove, California, USA, Sept. 1986

[DiGl87] Dittrich, K.R.; Gotthard, W.; Lockemann, P.C.: DAMOKLES-The Database System for the UNIBASE Software Engineering Environment. IEEE Database Engineering, March 1987

[DiKM85] Dittrich, K.R.; Kotz, A.M.; Mülle, J.A.: A Multilevel Approach to Design Database Systems and its Basic Mechanisms. Proc. IEEE COMPINT, Montreal, 1985

[DLPS85] Dadam, P.; Lum, V.; Praedel, U.; Schlageter, G.: Selective Deferred Index Maintenance and Concurrency Control in Integrated Information Systems. Proc. 11th Int. Conf. on Very Large Data Bases, Stockholm, 1985

[DüKe88] Dürr, M.; Kemper, A.: Transaction Control Mechanism for the Object Cache Interface or R^2D^2. Proc. 3rd Int. Conf. on Data and Knowledge Bases, Jerusalém, June 1988

[Eden82] Jessop, W.H. et al.: An Introduction to the Eden Transaction File System. Proc. 2nd IEEE Symposium on Reliability in Distributed Software and Database Systems, Pittsburgh, July 1982

[ElBa84] Elhardt, K.; Bayer, R.: A Database Cache for High Performance and Fast Restart in Database Systems. ACM Trans. on Database Systems, Vol. 9, No. 4, December 1984

[ErWa86] Erbe, R.; Walch, G.: Usage of the Application Program Interface of the Advanced In-formation Manager Prototype. Technical Note TN 86.03, IBM Heidelberg Scientific Center, Dec. 1986

[ErWa87] Erbe. R.; Walch, G.: An Application Program Interface for an NF^2 Database Language or How to Transfer Complex Object Data into an Application Program. Technical Report TR 87.04.003, IBM Heidelberg Scientific Center, April 1987

[Frei89] Freisleben, B.: Support for Reliability in an Object-Oriented Environment. Proc. 2nd Int. Workshop on Distribution and Objects, Karlsruhe, April 1989

[GaKi88l Garza, J.F.; Kim, W.: Transaction Management in an Object-Oriented Database System. Proc. ACM SIGMOD Int. Conf. on Management of Data, Chicago, June 1988

[Gray78] Gray, J.: Notes on Database Operating Systems. Lecture Notes on Computer Science, Vol. 60, Springer Verlag, New York, 1978

[Gray80] Gray, J.: A Transaction Model. Research Report RJ2895, IBM Research Laboratory, San Jose, California, 1980

[Gray81] Gray, J.N. et al.: The Recovery Manager of the System R Database Manager. In: ACM Comp. Surveys, Vol.13, No. 2, 1981

[HaKK81] Hatzopoulos, M.; Kollias, J.G.; Kollias, V.J.: The application of a Number of Differential Files to the Maintenance of Large Databases. Angewandte Informatik 1/81

[HaLo81] Haskin, R.L.; Lorie, R.A.: On Extending the Functions of a Relational Database System. Research Report RJ3182, IBM Research Laboratory, San Jose, California, 1981

[HäPR85] Härder, T.; Peinl, P.; Reuter, A.: Performance analysis of synchronization and recovery schemes. IEEE Database Engineering, 1985

[Härd87] Härder, T.: On Selected Performance Issues of Database Systems. Proc. 4th GI/ITG-Fachtagung Messung, Modellierung und Bewertung, Erlangen (W. Germany), Sept. 1987

→ [HäRe83] Härder, T.; Reuter, A.: Principles of Transaction-Oriented Database Recovery. ACM Computing Surveys, Vol.15, No. 4, December 1983

[HäRe85] Härder, T.; Reuter, A.: Architecture of Database Systems for Non-Standard Applications. In: Informatik Fachberichte, Vol. 94, Springer Verlag,1985 (in German)

[HäRo87] Härder,T.; Rothermel,K.: Concepts for Transaction Recovery in Nested Transactions. Proc. ACM SIGMOD Int. Conf. on Management of Data, San Francisco, May 1987

[HeSW75] Held, G.; Stonebraker, M.R.; Wong, E.: INGRES - A Relational Data Base System. In: Proc. AFIPS NCC, 1975

[HHMM88] Härder, T.; Hübel, Ch.; Meyer-Wegener, K.; Mitschang, B.: Processing and Transaction Concepts for Cooperation of Engineering Workstations and a Database Server. In: Data & Knowledge Engineering, North Holland, No. 3, 1988

[HMMS87] Härder, T.; Meyer-Wegener, K.; Mitschang, B.; Sikeler, A.: PRIMA - A DBMS Prototype Supporting Engineering Applications. Proc. 13th Int. Conf. on Very Large Data Bases, Brighton, England, 1987

[HüMi88] Hübel, C.; Mitschang, B.: Object Orientation within the PRIMA-NDBS. Proc. 2nd Int. Workshop on Object-Oriented Database Systems, Bad Münster am Stein-Eberburg (West Germany), Sept. 1988

[Ioc89a] Iochpe, C.: A DBMS Simulation Model based on both, a DBMS Kernel Architecture and a Set of Design Processing Models. Research Report, Forschungszentrum für Informatik, Karlsruhe, Feb. 1989

[Ioc89b] Iochpe, C.: Performance Analysis of Recovery Mechanisms in a Design Database System Kernel. Research Report, Forschungszentrum für Informatik, Karlsruhe, 1989 (in preparation)

[KARD88] Bültzingsloewen, G.v.; Iochpe, C.; Liedtke, R.-P.; Lockemann,P.C.: Two Level Transaction Management in a Multiprocessor Database Machine. In: Proc. 3rd Int. Conf. on Data and Knowledge Bases, Jerusalem, 1988

[KaWe84] Katz, R.H.; Weiss, S.: Design Transaction Management. Proc. 21th Design Automation Conference, June 1984

[Kelt88] Kelter, U.: Transaction Concepts for Non-Standard Database Systems. Informationstechnik (it), No. 30, January 1988 (in Germany)

[KeWa87] Kemper, A.; Wallrath, M.: An Object-Oriented Application Program Interface to an Engineering Database System. Research Report No.30/87, Faculty of Computer Science, University of Karlsruhe, Sept. 1987 (in German)

[KeWa88] Kemper, A.; Wallrath, M.: A Uniform Concept for Storing and Manipulating Engineering Objects. Proc. 2nd Int. Workshop on Object-Oriented Database Systems, Bad Münster am Stein-Eberburg (West Germany), Sept. 1988

[KHED89] Küspert, K.; Hermann, H.; Erbe, R.; Dadam, P.: The Recovery Manager of the Advanced Information Management Prototype. Proc. 7th National Reliability Engineering Conference - Reliability'89, Brighton (UK), 1989

[KLMP84] Kim, W.; Lorie, R.A.; McNabb, D.; Plouffe, W.: A Transaction Mechanism for Engineering Design Databases. Proc. 10th Int. Conf. on Very Large Data Bases, Singapore, 1984

[Kohl81] Kohler, W.H.: A Survey of Techniques for Synchronization and Recovery in Decentralized Computer Systems. ACM Computing Surveys, Vol. 13, No. 2, June 1981

[KoKB87] Korth, H.F.; Kim, W.; Bancilhon, F.: On Long-Duration CAD Transactions. Information Science, 1987

[Kotz88] Kotz, A.M.; Dittrich, K.R.; Mülle, J.A.: Supporting Semantic Rules by a Generalized Event/Trigger Mechanism. Proc. of the Int. Conf. on Extending Database Technology, Venice, March 1988

[KSUW85] Klahold, P.; Schlageter, G.; Unland, R.; Wilkes, W.: A Transaction Model Supporting Complex Applications in Integrated Information Systems. In: Proc. ACM SIGMOD Int. Conf. on Management of Data, Austin, Texas, 1985

[KüDG87] Küspert, K.; Dadam, P.; Günauer, J.: Cooperative Object Buffer Management in the Advanced Information Management Prototype. Proc. 13th Int. Conf. on Very Large Data Bases, Brighton, 1987

[LaSt79] Lampson, B.; Sturgis, H.: Crash Recovery in a Distributed Data Storage System. Xerox Research Memo, Xerox PARC, April 1979

[LCJS87] Liskov, B.; Curtis, D.; Johnson, P.; Scheifler, R.: Implementation of Argus. Proc 11th ACM Symposium on Operating Systems Principles, Austin, November 1987

[LeRV87] Lecluse, C.; Richard, P.; Velez, F.: O2, An Object-Oriented Data Model. Proc. Workshop on Database Programming Languages, Roscoff, France, Sept. 1987

[Lind79] Lindsay, B.G. et al.: Notes on Distributed Databases. IBM Research Report RJ2571, IBM Research Loboratory, San Jose, California (USA), 1979

[Lock85] Lockemann, P.C. et al.: Requirements of Engineering Applications on Database Systems. Proc. Workshop on Database Systems for Office, Engineering, and Scientific Applications, Springer Verlag, Karlsruhe, 1985 (in German)

[LoPl83] Lorie, R.A.; Plouffe, W.: Complex Objects and Their Use in Design Transactions. Engineering Design Applications Proc. of the Annual Meeting: Database Week, San Jose, California, May 1983

[LoSc88] Lockemann, P.C.; Schmidt, J.W.: Database Handbook. Springer Verlag, 1988 (in German)

[Lori77] Lorie, R.A.: Physical Integrity in a Large Segmented Database. ACM Trans. on Database Systems, Vol. 2, No. 1, March 1977

[Lync83] Lynch, N.A.: Multilevel Atomicity - A New Correctness Criterion for Database Concurrency Control. ACM Trans. on Database Systems, Vol. 8, No. 4, 1983

[Meye86] Meyer-Wegener, K.: Transaction Systems: An investigation of their functionality, implementation strategies, and performance. Ph.D. dissertation, University of Kaiserslautern, 1986 (in German)

[Mits87] Mitschang, B.: MAD - A Data Model for the Kernel of a Non-Standard Database System. Proc. Workshop on Database Systems for Office, Engineering, and Scientific Applications, Springer Verlag, Darmstadt, 1987 (in German)

[MoAb86] Garcia-Molina,H.; Abbott,R.K.: Reliable Distributed Database Management. Research Report CS-TR-047-86, Department of Computer Science, Princeton University, August 1986

[Moss81] Moss, J.E.: Nested Transactions: An Approach to Reliable Distributed Computing. Ph.D. dissertation, Dept. of Electrical Engineering and Computer Science, MIT,1981

[Moss82] Moss, J.E.: Nested Transactions and Reliable Distributed Computing. Proc. 2nd IEEE Symposium on Reliability in Distributed Software and Database Systems, Pittsburgh, July 1982

[Moss87] Moss, J.E.: Log-Based Recovery for Nested Transactions. Proc. 13th Int. Conf. on Very Large Data Bases, Brighton, 1987

[MuMP83] Mueller, E.T.; Moore, J.D.; Popek, G.J.: A Nested Transaction Mechanism for LOCUS. Proc. 9th ACM Symposium on Operating Systems Principles, Bretton Woods, October 1983

[PaJF89] Pathak, G.; Joseph, J.; Ford, S.: Object eXchange Service for an Object-Oriented Database System. Proc. 5th Int. Conf. on Data Engineering, Los Angeles, Feb. 1989

[Paul87] Paul, H.-B. et al.: Architecture and Implementation of the Darmstadt Database Kernel System. Proc. ACM SIGMOD Int. Conf. on Management of Data, 1987

[Pist86] Pistor, P.: Andersen, F.: Designing a Generalized NF^2 Data Model with SQL-type Language Interface. Proc. 12th Int. Conf. on Very Large Data Bases, 1986

[PROF85] Nett, E. et al.: PROFEMO - Design and Implementation of a Fault Tolerant Distributed System Architecture. GMD-Studien Nr 100, Geselschaft für Mathematik und Datenverarbeitung MBH, Sankt Augustin (W. Germany), 1985

[PSSW87] Paul, H.-B; Schek, H.-J; Söder, A.; Weikum, G.: Supporting the Office Filing Service by a Database Kernel System. Proc. GI Conf. Data Base Systems for Office, Engineering, and Scientific Applications, Darmstadt, 1987

[Rand78] Randell, B. et al.: State Restoration in Distributed Systems. Proc. of the 8th FTCS, Toulouse (France), 1978

[Reüt80] Reuter, A.: Recovery Architecture for Database Systems. Proc.of the Int. Computing Symposium, London, Westbury House, 1980

[Reut84] Reuter, A.: Performance Analysis of Recovery Techniques. ACM Trans. on Database Systems, Vol. 9, No. 4, December 1984

[Ries89] Riess, H.: A Recovery Mechanism for a Main Memory Resident Object Cache. Graduation Work Report (Diplomarbeit), Informatic Faculty, University of Karlsruhe, July 1989 (in German)

[Roth85] Rothermel, K.: Communication Support for Distributed Database Systems. Proc. GI-NTG Conf. on Communication in Distributed Systems, 1985

[RoSt87] Rowe, L.A.; Stonebraker, M.R.: The POSTGRES Data Model. Proc.13th Int. Conf. on Very Large Data Bases, Brighton, 1987

[Rowe86] Rowe, L.A.: A Shared Object Hierarchy. Proc. Int. Workshop on Object-Oriented DBS, Pacific Grove, California, USA, Sept. 1986

[Schm88] Schmid, H.: The Development of a Non-Standard Database System Simulation Model. Individual Study (Studienarbeit), Fakultät für Informatik, Universität Karlsruhe, April 1989 (in German)

[ScWe86] Schek, H.-J.; Weikum, G.: DASDBS: Concepts and Architecture of a Database System for Advanced Applications. DVSI-86-TI, Technical College of Darmstadt, 1986

[SeLo76] Severance, D.G.; Lohman, G.M.: Differential Files: Their Application to the Maintenance of Large Databases. ACM Trans. on Database Systems, Vol.1, No. 3, Sept. 1976

[Senk73] Senko, M.E. et al.: Data Structures and Access in Data Base Systems. IBM Journal, No.12, 1973

[Ston87] Stonebraker, M.: The Design of the POSTGRES Storage System. Proc. 13th Int. Conf. on Very Large Data Bases, Brighton, 1987

[StRo84] Stonebraker, M.; Rowe, L.A.: Database Portals: A New Application Program Interface. Proc.10th Int. Conf. on Very Large Data Bases, Singapore, 1984

[StRo86] Stonebraker, M.; Rowe, L.A.: The Design of POSTGRES. ACM SIGMOD Int. Conference on Management of Data, Washington, May 1986

[Trai83] Traiger, I.L.: Trends in System Aspects of Database Management. ICOD-2, Cambridge (UK), 1983

[Verh78] Verhofstad, J.S.M.: Recovery Techniques for Database Systems. ACM Computing Surveys, Vol. 10, No. 2, June 1978

[Verh79] Verhofstad, J.S.M.: Recovery Based on Types. Data Base Architectures, Bracchi and Nijssen eds., North-Holland Publishing Company - IFIP 1979

[Wei87a] Weikum, G.: Principles and Realization Strategies of Multi-Level Transaction Management. Research Report, Technical University of Darmstadt 1987

[Wei87b] Weikum, G.: Transaction Management in Database Systems with Layered Architecture. Ph.D. dissertation, Technical University of Darmstadt, 1987 (in German)

[WeKa84] Weiss, S.; Katz, R.H.: Recovery of In-Memory Data Structures for Interactive Update Applications. Proc. IEEE COMPCON, 1984

[WeNP87] Weikum, G.; Neumann, B; Paul, H.-B.: Concept and Realization of a Set-Oriented Page-Layer for Efficient Access to Complex Objects, Proc. GI Conf. Database Systems for Office Automation, Engineering, and Scientific Applications, Darmstadt 1987

Lebenslauf

Cirano Iochpe

04.01.1956	geboren in Porto Alegre, Brasilien, als Sohn des Buchhalters Isaac Iochpe und seiner Frau Frida geb. Zatz
1963 - 1974	Besuch der Grundschule und des Gymnasium ´Colégio Israelita Brasileiro´ in Porto Alegre
1975	Aufnahmeprüfung bei der Bundesuniversität UFRGS (Universidade Federal do Rio Grande do Sul), Fachrichtung Elektrotechnik
1975 - 1978	Studium an der UFRGS, Fachrichtung Elektrotechnik
1977 - 1981	Programmierer im Rechenzentrum der UFRGS
1978	Aufnahmeprüfung bei der Bundesuniversität UFRGS, Fachrichtung Informatik
1978 - 1979	Studium an der UFRGS, Fachrichtung Informatik
1980	Fortbildungskurs für Software-Engineering des Rechenzentrums der Bundesuniversität UFRGS
1981 - 1983	´Mestrado´ in Ciência da Computação (Master in Computer Science) an der UFRGS
1982	Systemanalytiker im Rechenzentrum der UFRGS
1983 - 1985	Wissenschaftlicher Mitarbeiter am Pós-Graduação em Ciência da Computação der UFRGS
1985 - 1989	Beurlaubt zur Promotion an der Fakultät für Informatik der Universität Karlsruhe als DAAD-Stipendiat
1986 - heute	Wissenschaftlicher Angestellter an der UFRGS, als ´Assistente´ (zweite Stufe eines brasilianischen Hochschullehrers) in der Fachrichtung Informatik