Transaction and Version Management in Object-Oriented Database Management Systems for Collaborative Engineering Applications

by

Shamim Ahmed

B.S., IIT Kharagpur, India (1987)
M.S., Case Western Reserve University, Cleveland, OH (1989)

Submitted to the Department of Civil Engineering
in partial fulfillment of the requirements for the degree of

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

January 1991

© Massachusetts Institute of Technology 1991. All rights reserved.
Transaction and Version Management in Object-Oriented Database Management Systems for Collaborative Engineering Applications

by

Shamim Ahmed

Submitted to the Department of Civil Engineering on January 1991, in partial fulfillment of the requirements for the degree of Master of Science

Abstract

A large computer-aided-design or engineering (CAD/CAE) project (such as aircraft design) typically involves a group of designers working cooperatively on distributed workstations to complete a complex design by closely interacting among themselves and dynamically sharing design data and information. Such design environments necessitate powerful data modeling and versioning, sharing and management tools, network communication protocols and a flexible framework for concurrency management of highly interleaved and interactive transactions. This study discusses the essential features of such a transaction management framework. It forms a integral component of an ongoing effort at IESL, called DICE (Distributed and Integrated Computer-aided Engineering Environment), which consists of a set of object-oriented tools aimed at addressing coordination and communication problems in collaborative engineering.

The conventional model of database transactions is based on the notions of serializability and atomicity. Although this may be adequate for traditional data processing applications, it is not appropriate for CAD applications that are developed for distributed environments and involve several users in a collaborative activity. CAD transactions are usually of long duration, interleaved and represent interactive modifications to a complex design. Serializability is often too limiting a criteria for such cooperative CAD work, and may result in intolerably long waits, reduced concurrency or undoing of a significant amount of work.

Special considerations are therefore necessary for the design of a more flexible and efficient transaction management system which allows a group of cooperating transactions to arrive at a complex design without being forced to wait over a long duration, and enables collaboration among design groups.

Object-oriented database management systems (OODBMS) provide a powerful medium for modeling, coordination, storage and manipulation of engineering information, and also present an opportunity to provide greater concurrency than more traditional approaches allow.

This study presents a transaction management model for an object-oriented database environment which supports a high degree of parallelism while preserving database consistency. Some of its novel features include facilities such as flexible, commu-
nicate lock types, inter-client communication protocols, conflict handling, grouped and shared transactions, database partitioning into local and global shared areas, object registration and checkpointing, version management of data, and alternative concurrency control strategies. It also establishes protocols for transaction grouping, and consistency maintenance on the basis of application semantics and encapsulation of nonserializable data sharing in local databases rather than by the notion of global database consistency. Finally, it presents strategies for implementing these features using a commercial OODBMS environment (ONTOS).

Thesis Supervisor: Duvvuru Sriram
Title: Assistant Professor
Acknowledgments

The author would like to acknowledge:

- Professor Duvuru Sriram, for his guidance, active encouragement and help in all matters;
- Professor Robert Logcher, for discussions, useful comments and insight in the DICE meetings;
- Professor John Williams for inspiration, helpful discussions and his friendly disposition;
- Mauricio Daher, system manager at IESL for extensive help with the computers and software;
- Joan McCusker and Georgia Taylor, for administrative support;
- Jacqueline Fischer, presently at Oracle Corp., for her warm friendship and emotional support;
- Albert Wong, for helpful discussions and assistance in all matters;
- Ashutosh Gupta, for managing the 2.157 class project single-handedly while I was away;
- Feniosky Pena, for his jovial friendship, and for cooking those great dinners;
- Pavan Bakeri, for our merry regular after-midnight laugh and joke sessions;
- John Tom for his conceptual philosophies and for getting me started with Latex;
- All other friends and associates who helped to warm things up in this cold environment;
- Finally, and most importantly, my parents for standing by me through everything.
Contents

1 Introduction ............................................. 10
1.1 Introduction ........................................ 10
1.2 DICE ................................................. 11
1.2.1 Introduction ..................................... 11
1.2.2 DICE System Architecture ......................... 12
1.2.3 DICE Blackboard: Object-Oriented Database ....... 14
1.3 An Example of Collaborative Development ............ 17
1.4 Requirements of Transaction Management Framework in DICE 27
1.5 Thesis Outline ....................................... 30

2 Background ............................................ 31
2.1 Introduction - OODBMS ............................... 31
2.1.1 Characteristics of OODBMS ...................... 31
2.2 Requirements of a Database System for Collaborative Engineering Applications ............... 33
2.3 Advantages of OODBMS over RDBMS for Engineering Applications .................. 36
2.4 Summary ............................................. 39

3 The ONTOS Commercial OODBMS ....................... 40
3.1 Introduction ......................................... 40
3.2 ONTOS System Architecture .......................... 41
3.3 Basic Data Modeling Using ONTOS .................... 41
3.4 Composite Object Facilities .......................... 43
3.5 Dynamic Schema Evolution Facilities ................ 43
3.5.1 Taxonomy of Schema Evolution .................... 43
3.5.2 Schema Evolution in ONTOS ...................... 47
3.6 Object Storage Management .......................... 47
3.7 Basic Transaction Management Features ................ 50
3.8 Client/Object/Transaction Data Management .......... 50
3.9 Lock Management .................................... 50
3.10 Communication and Change Notification .............. 51
3.11 Version Management .................................. 51
3.12 Concurrency Management ............................ 51
3.13 Conflict Handling .................................... 52
3.14 Facilities for Cooperative Engineering .............. 52
3.15 Query Management ..................................................... 53
3.16 Database Authorization and Security .............................. 53
3.17 Multimedia Data Management ..................................... 53
3.18 User Interface ......................................................... 53
3.19 Summary ............................................................... 54

4 A Transaction Management Framework for Collaborative CAD 55
   4.1 Introduction .......................................................... 55
   4.2 Transaction Management Functionalities in a DBMS .......... 56
   4.3 A Brief Review of Traditional Transaction Management Frameworks and their Limitations for CAD .................................. 58
   4.4 A Model for a Collaborative CAD Environment .............. 60
   4.5 Transaction Management Functionalities for Collaborative CAD ........................................... 67
      4.5.1 Interactive Transaction Framework ...................... 67
      4.5.2 Client, Data, and Transaction Record ................. 67
      4.5.3 Lock Management ............................................ 68
      4.5.4 Deadlock Management ..................................... 73
      4.5.5 Communication .............................................. 74
      4.5.6 Version Management ....................................... 76
      4.5.7 Conflict Handling .......................................... 90
      4.5.8 Transaction Nesting ....................................... 92
      4.5.9 Transaction Groups ....................................... 93
      4.5.10 Shared Transactions ..................................... 100
      4.5.11 Transaction Checkpointing and Object Registration .......... 103
      4.5.12 Concurrency Management and Data Consistency ........ 104
   4.6 Summary ............................................................. 105

5 Implementation Issues .................................................. 107
   5.1 Introduction ........................................................ 107
   5.2 A Dynamic Transaction Framework .............................. 107
      5.2.1 Dynamic Linking ........................................... 108
      5.2.2 Implementation details ................................... 108
      5.2.3 Examples of Dynamic Linking ............................ 110
   5.3 Client, Object and Transaction Record ...................... 114
   5.4 Communication Protocols ...................................... 116
   5.5 Update Notification ............................................. 116
   5.6 Interactive Lock Management ................................ 116
   5.7 Transaction Conflict Management .............................. 116
   5.8 Version Management ............................................. 117
   5.9 Version Configurations ......................................... 118
      5.9.1 Versions of Composite Objects ........................ 118
      5.9.2 Versions of Aggregates .................................. 118
   5.10 Database Partitions ............................................... 119
   5.11 Implementation Problems Using ONTOS ...................... 123
   5.12 Summary .......................................................... 123
6 Conclusion

6.1 Summary ................................................. 125
6.2 Current and Future Work ................................. 126

A Transaction Management Classes ...................... 129
A.1 Class root ................................................. 129
A.2 Class User ................................................ 130
A.3 Class Transaction ....................................... 131
A.4 Class Dependency_dictionary ........................... 132
A.5 Class History_dictionary ............................... 132
A.6 Class version_obj ........................................ 133
A.7 Class Version_set ....................................... 133
A.8 Class Lock_dictionary ................................... 134
A.9 Versionable Collection Objects ....................... 135
# List of Figures

1-1 DICE coordination framework ........................................ 13
1-2 A conceptual view of DICE for design and construction .......... 15
1-3 Layered architecture for DICE BB .................................... 16
1-4 Agents involved in building design .................................. 18
1-5 Building components .................................................... 19
1-6 DICE framework for building design .................................. 20
1-7 Illustrative database partitioning for design ......................... 21
1-8 Design steps in building design (simplified) ......................... 23

3-1 ONTOS distributed client-server architecture ....................... 42
3-2 Composite objects in a Car ............................................ 44
3-3 Object activation and deactivation ................................... 48
3-4 Granularity of activation and deactivation functions .............. 49

4-1 Transaction management modules in a DBMS ....................... 57
4-2 Task discretization in a design project .............................. 62
4-3 Task discretization in design of building ............................ 63
4-4 Model of CAD transactions showing transaction nesting and grouping .................................................. 64
4-5 Database partitioning for collaborative CAD ......................... 66
4-6 An example of non-serialized "correct" schedule in building design .................................................. 72
4-7 Example of enhanced concurrency due to versioning of objects .......... 78
4-8 Linear and parallel versions ............................................ 80
4-9 Check-out and check-in (single user) ................................ 81
4-10 Check-out and check-in (multiple users) ............................. 82
4-11 Version management of classes ....................................... 84
4-12 Composite object configurations ..................................... 86
4-13 Example of composite object versioning showing change in significant component .................................................. 87
4-14 Example of composite object versioning showing change in isolated component .................................................. 88
4-15 Motivating example for transaction grouping ........................ 95
4-16 Solution by transaction grouping ..................................... 96
4-17 Transaction group tree .................................................. 97
4-18 Example of grouped transaction in building design .................. 99
4-19 Cooperative interaction in a shared transaction ..................... 101
4-20 Example of type-based concurrency management ..................... 106
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5-1</td>
<td>Dynamic linking</td>
<td>109</td>
</tr>
<tr>
<td>5-2</td>
<td>Hierarchy of defined base classes</td>
<td>115</td>
</tr>
<tr>
<td>5-3</td>
<td>Desired setup for versionable collection classes</td>
<td>120</td>
</tr>
<tr>
<td>5-4</td>
<td>Version management of collections with versionable components</td>
<td>121</td>
</tr>
<tr>
<td>5-5</td>
<td>Version management of collections with non-versionable components</td>
<td>122</td>
</tr>
<tr>
<td>5-6</td>
<td>Problem of recording simultaneous lock acquisitions on an object</td>
<td>124</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Introduction

A large computer-aided-design or engineering (CAD/CAE) project (such as aircraft design) typically involves a group of designers working cooperatively on distributed workstations to complete a complex design by closely interacting among themselves and dynamically sharing design data and information. Such design environments necessitate powerful data modeling, sharing and management tools, network communication protocols and a flexible framework for concurrency management of highly interleaved and interactive transactions. This study presents the essential features of such a transaction management framework. It forms a integral component of an ongoing effort at MIT, called DICE (Distributed and Integrated Computer-aided Engineering Environment) [44], which consists of a set of object-oriented tools aimed at addressing coordination and communication problems in collaborative engineering.

Special considerations are therefore necessary for the design of a more flexible and efficient transaction management system which allows a group of cooperating transactions to arrive at a complex design without being forced to wait over a long duration, and enables collaboration among design groups.

Object-oriented database management systems (OODBMS) provide a powerful medium for modeling, coordination, storage and manipulation of engineering information, and due to the greater semantic content of the database, also present an opportunity to provide greater concurrency than more traditional approaches allow [45]. An OODBMS is a repository of objects which encapsulate in the data structure, both data and behavior in the form of methods. Objects interact with each other by invocation of each others' methods. All changes to object data take place through execution of one or more methods. This enables the data (or objects) to perform intelligent operations when accessed, changed or updated. A change in the state of an object may automatically trigger several meaningful operations. Thus, in an object-oriented approach, transactions consists of a set of operations on design objects which reason about their domain, and have more semantics than just read or write. It may be shown that though concurrent transactions may not be serializable, "correctness" can be enforced on the basis of application or type-defined semantics rather than by the no-read-write-conflict approach.
OODBMS therefore provide a better environment for collaborative engineering design management than relational database management systems (RDBMS) do. The present work studies how object-oriented tools may be used to build transaction management frameworks for collaborative engineering CAD such as proposed for DICE.

This chapter has been organized as follows. First, we provide a brief overview of DICE and its system architecture. Next, we provide an engineering example illustrating DICE concepts. This is followed by identification of the transaction management requirements for supporting such a framework.

1.2 DICE

1.2.1 Introduction

Engineering projects generally involve a large number of components and interaction of multiple technologies. The components included in the product are decided in an iterative design process. In each iteration, interfaces and interface conditions among these components are designed with slack to account for potential variations created when the components and interface values become better known. Iteration proceeds towards greater detail. This multi-faceted nature of engineering problems demands considerable coordination between various participants. Lack of coordination may result in several undesirable effects, which may result in delay, design flaws and loss of productivity.

The DICE framework has been envisioned as a computer-aided medium for solving some of these problems of coordination. As described in [44] the major objectives of DICE are to:

1. Facilitate effective coordination and communication in various disciplines involved in engineering.

2. Capture the process by which individual designers make decisions, that is, what information was used, by who, how it was used and what it was used to create.

3. Forecast the implications of design decisions on manufacturing and construction.

4. Provide designers interactively with detailed manufacturing process or construction planning.

5. Develop intelligent interfaces for automation.

A framework like DICE will significantly improve productivity by

- reducing error in design;

- providing more detailed design;

- providing better manufacturing and construction planning;
• allowing easier recognition of design and manufacturing (construction) problems;

• using manufacturability criteria throughout design; and

• advancing automation.

1.2.2 DICE System Architecture

DICE can be envisioned as a network of computers and users, where the communication and coordination is achieved through a global database and a control mechanism (Figure 1-1).

DICE consists of a Blackboard (global database), several Knowledge Modules, and a Control Mechanism. These terms are clarified below [44].

1. Blackboard. The Blackboard is the medium through which all communication takes place. The Blackboard in DICE is divided into three partitions: Solution, Negotiation, and Coordination Blackboards. The Solution Blackboard partition contains the design and construction information generated by various Knowledge Modules; this solution is normally referred to as the Object-Hierarchy. The Negotiation Blackboard partition consists of the negotiation trace between various engineers taking part in the design and manufacturing (construction) process. The Coordination Blackboard partition contains the information needed for the coordination of various Knowledge Modules.

2. Knowledge Module. Each Knowledge Module (KM) can be viewed either as: a knowledge based expert system (KBES), developed for solving individual design and construction related tasks, or a CAD tool, such as a database structure, i.e., a specific database, an analysis program, etc., or an user of a computer, or a combination of the above. A KBES could be viewed as an aggregation of Knowledge Sources (KSs). Each KS is an independent chunk of knowledge, represented either as rules or objects. In DICE, the Knowledge Modules are grouped into the following categories: Strategy, Specialist, Critic, and Quantitative. The Strategy KMs help the Control Mechanism in the coordination and communication process. The Specialist KMs perform individual specialized tasks of the design and construction process. The Critic KMs check various aspects of the design process, while the Quantitative KMs are mostly algorithmic CAD tools. The data representation (or language) used in a KM may be different from that used in the BB. Hence, each KM is provided with an interface module which translates the data from the KM to the BB and vice versa.

3. Control Mechanism. The Control Mechanism performs two tasks: 1) evaluate and propagate implications of actions taken by a particular KM; and 2) assist in the negotiation process. This control is achieved through the object-oriented nature of the Blackboard and a Strategic KM. One of the major and unique difference between DICE and other Blackboard systems is that DICE's Blackboard
Figure 1-1: DICE coordination framework
is more than a static repository of data; DICE's Blackboard is an intelligent database, with objects responding to different types of messages.

A conceptual view of DICE for design and construction is shown in Figure 1-2; a detailed description of a LISP-based implementation of DICE is provided in [44]. In the DICE framework, any of the KMs can make changes or request information from the Blackboard; requests for information are logged with the objects representing the information, and changes to the Blackboard may initiate either of the two actions: finding the implications and notifying various KMs, and entering into a negotiation process, if two or more KMs suggest conflicting changes.

1.2.3 DICE Blackboard: Object-Oriented Database

The Blackboard is being implemented as an object-oriented database. Encapsulation of data and procedures in objects, inheritance of properties, and abstraction of data are some of the characteristics that makes the object-oriented database an ideal model for encoding engineering information; a detailed discussion on the relevance of object-oriented database management systems (OODBMS) for engineering applications is provided in Chapter 2. Solution BB (SBB) and Coordination BB (COORDBB) partitions are being implemented as a layered object-oriented database, as shown in Figure 1-3. The various layers are described briefly below.

1. **Physical Layer.** Data resides in the form of bits on an appropriate storage medium (e.g., magnetic, optical, video disks).

2. **Storage Layer.** Objects are assigned unique identifiers, which are mapped into appropriate areas in the Physical Layer.

3. **Controller Layer.** Grouping of objects, allocation and de-allocation of object buffers, and other storage control activities are achieved at this layer.

4. **Object-base Layer.** Object definition, modification, and other associated activities are included here.

5. **Version Layer.** Versions of objects help to keep track of the design evolution and also enhances parallelism of design activities. Various version management facilities are encoded at this layer.

6. **Transaction Layer.** Transaction management layer is responsible for maintaining database integrity, while allowing execution of multiple concurrent transactions by various engineers. This layer supports a transaction framework for collaborative engineering applications.

The present study is concerned principally with the object-base, version, transaction and control layers of the DICE framework, with special emphasis on data versioning and transaction control.
Figure 1-2: A conceptual view of DICE for design and construction

Note: User/CAD interface is optional in KMs. Also messages through Interface Def. are not shown.
Figure 1-3: Layered architecture for DICE BB
1.3 An Example of Collaborative Development

Most large engineering projects, such as aircraft design, design of buildings, CASE, etc., are collaborative in nature, i.e., they involve several participants in interaction for the achievement of a common goal.

We will introduce our proposed framework for transaction management in DICE with a simple example. As an illustration of collaborative product development, consider the example of building design and construction. The design of a building involves teams of several designers, different technologies and components. There is considerable necessity for controlled interaction and cooperation between different design groups for the successful completion of the design task. Some of the design technologies and agents are shown in Figure 1-4. These include (among others): project manager, architect, structural engineer, geotechnical engineer, HVAC (heating, ventilation and air-conditioning) engineer, electrical engineer, plumbing and sanitary engineer, fabricator, contractor, owner, etc. We will consider a very simplified scenario involved in the design of the of a small building.

The various components of a building are shown in Figure 1-5. It consists of a superstructure and a substructure. The architect is responsible for designing the skeletal plan of the building, positions of beams, columns, layout of rooms, hallways, and interior design. The structural engineer takes specifications from the architect to design the superstructural elements of the building, such as beams, columns, slabs, connections, stairways, joints, etc. The geotechnical engineer takes the estimate of column loads from the structural engineer to design the substructure of the building. This includes components like the footing, foundation, water-proofing and water-retaining structures, etc. The house is modeled as a composite object [20, 43] in the DICE object-oriented database (DICE BB). The House is a containing object which is composed of component objects such as Superstructure and Substructure, which are themselves composed of several other objects such as Beam, Column, etc. (Figure 1-5).

The principal agents involved in our simplified scenario are the architect (A), structural (S) and geotechnical (G) engineers. As envisioned in the DICE framework, these agents are working on individual client workstations on a network with the database (or blackboard with control mechanisms) residing on a server machine (Figure 1-6).

Since collaborative engineering entails data and information sharing, it necessitates partitioning of the database into local shared areas. The database architecture for our example (at some stage of the design process) is shown in Figure 1-7. Each designer has his/her own private dataspace for doing work that is not accessible to others. When designers need to cooperate, they work on local shared dataspaces that are derived from the global database (or subdivisions thereof) that are read/write accessible to all of them.

In this example, A, S and G are working as a design group and share a local dataspace which at startup, is either empty, or contains a copy of all the relevant objects (required for design) taken from a consistent database. All changes made to these objects are visible only in the scope of the shared database, so intermittent
Figure 1-4: Agents involved in building design
Figure 1.5: Building components
Figure 1-6: DICE framework for building design
Figure 1-7: Illustrative database partitioning for design
changes during the group's design effort do not affect other designers not concerned with the group's activities. When all designers are satisfied with their design, appropriate objects in the shared database will be released to the global database (or to the parent databases from which the local database was derived), so that the new objects can be shared with others in the group.

The following represents the steps and interactions by which the design would normally proceed (Figure 1-8). It is assumed that each designer is assisted by a set of automated analysis and design tools (represented by knowledge modules in the DICE framework), such as analysis and design packages.
Figure 1-8: Design steps in building design (simplified)
Shared Database State

Architect, A  Structural engg., S  Geotechnical engg., G

Make changes to initial specifications, post changes

 notifies

Check out changed components, check to see if design needs to be changed

Post design changes if any

 notifies

Check out new data for possible redesign

A completes final layout, posts final design

 notifies

Check out completed configuration for structural integrity, complete final design, post

 notifies

Check for redesign, complete final design, post foundation design

Design completed and agreed upon by all members after possible negotiation/constraint satisfaction, etc.

Check-out version H2 to global or parent database for sharing with others

Figure 1-8: Design steps in building design (continued)
• The architect (A) designs the skeletal layout of the building, positions of columns, beams and preliminary sizing of these components. This information can get the structural engineer started with his/her design, so A posts these results to the database. A may include dimensional or other constraints on the various building component parameters that S may have to abide by. In the meantime, the architect can continue with details of internal layout, such as walls, partitions, etc.

• The structural engineer S is notified of the posting, retrieves the House from the database and proceeds with the preliminary design, such as estimation of live and dead loads, sizing of the components, etc. At this stage, an estimate of the loads on the columns are known, so the geotechnical engineer can start with preliminary substructure design. S therefore posts his preliminary design to the shared database. A then continues with detailed structural analysis and design.

• The geotechnical engineer is notified of the posting. S/he retrieves the appropriate components (e.g., the column objects) from the database and proceeds with preliminary foundation design, such as the distribution area required for pressure dissipation, type of foundation required, etc.

• As S proceeds with detailed top-down structural design, the column loads become better known. After the design of each floor (from top down), S posts results to the database, which gives G notifications of better estimates of loads on the foundation. G may then proceed with the detailed design accordingly or refine previous previous preliminary design. By the time S is finished with detailed design of the superstructure, G will have finished a considerable amount of work on the substructure as well.

• As A proceeds with detailed layout, he/she may find it necessary to make modifications to previous designs. A posts these changes to the database, so that S is notified immediately. S can immediately look into the changes made to check whether it necessitates changes in his/her design. If so, changes can be effected, and the implications passed on to G.

It may be noted that whenever changes are made to existing data (or objects) in the database, the old data is not overwritten, but a new version of the object is created (version management is discussed in Chapter 4). This preserves the design history and enables the design to be restarted from a given specified state.

• When the architectural layout is complete, A posts the entire layout to the database. This configuration is retrieved by S to check for structural integrity of the superstructure. If it passes the check, any changes in column loads is passed on to G who refines the substructure design.

• In the event that there are anomalies or disagreement between designers, there is necessity for closer scrutiny of the design, greater degree of interaction, and
possibly negotiation for an agreeable design. For example, A and S do not agree on the dimensions of a beam, or the design requires dimensions that are beyond the range specified by A. In such cases, A and S may form another nested design group between themselves with a smaller shared dataspace containing only the relevant objects, and resolve the conflict by negotiation. If active experimental interaction is required, A and S may participate in a shared transaction. The concept of a shared transaction will be explained in Chapter 4; it suffices here to say that A and S may initiate a common transaction between themselves rather than having to communicate across transaction boundaries.

- When all design components have been agreed upon, the appropriate objects are checked out into the global database, where a new version of these design objects are created. These new designs may then be shared with other members, such as HVAC and electrical engineers. Typically, the architectural, structural and HVAC engineers would then form another design group to complete the HVAC design of the building, and so forth.

The following points of interest may be noted regarding the methodology suggested above:

1. It may be noticed that the above scheme allows engineers in different disciplines to proceed with their work in parallel, although the preceding design group has not completed its task. This is because the amount of information that is necessary to get other designers in the group started is released early, and is communicated to those concerned. This significantly increases the concurrency of design effort.

2. The system maintains records of data use by various designers and establishes dependencies between data and clients. Thus changes to data or objects result in notification to the appropriate designers who have used or accessed the data before. For example, a change in the Column object by S would immediately notify G. Thus the designer does not have to look out for changes, his/her attention is drawn automatically when appropriate pieces of data are changed.

3. With several designers collaborating in design effort, and changing data interactively, it is necessary to embed validity constraints on design data in the database. Whenever these constraints are violated by an update, the designer(s) concerned are warned of the violation, so that they may backtrack and redo the design appropriately. Constraint violations also need to be logged, and reported to those who have set them. For example, A may set a constraint such that the perimeter of a column should not exceed n units. When S posts a column design, this constraint is checked for validity.

The above framework provides a versatile and flexible platform to enable and coordinate collaborative design in most CAD disciplines. What are the underlying transaction features and technologies that are required to sustain the above framework? Several requirements like communication protocols and database structuring
are self evident. These and others are enumerated in the following section and discussed in detail in Chapter 4.

1.4 Requirements of Transaction Management Framework in DICE

A collaborative engineering environment, as illustrated in the previous example, necessitates a powerful and flexible transaction management framework to coordinate the concurrent activities of multiple users at a time. Most traditional database management systems do not support facilities for such collaborative work. Some of the principal requirements of the DICE transaction management framework are enumerated below.

1. A dynamic transaction framework. Engineering CAD transactions are highly interactive, iterative, and interleaved. Since design progresses through experimentation and incremental refinement, and since the nature of one transaction may depend on the result of other concurrent transactions, transactions need to be defined interpretively and dynamically at run-time. This would allow the designers to develop their applications incrementally and incorporate changes without complete application shutdown.

2. Communication facilities. Effective inter-client communication is of paramount importance in cooperative CAD work since it facilitates active exchange of information, better synchronization of work across design interfaces and greater concurrency, all of which are essential for collaborative development. The following types of communication modes are necessary:

   (a) Lock communication modes, which would send notification to appropriate clients regarding lock requests, attainments, releases, conflicts, queuing and denials.

   (b) Update communication modes, which would send messages to appropriate users regarding changes to specific data made by other users. This keeps all designers aware of the state of the database, and of each others' work.

   (c) Conflict communication modes, which would notify appropriate users in the events of conflicts such as locking, data access, commitment, or constraint violation conflicts. The users may then interact between themselves to resolve the conflict most reasonably. The idea is to avoid arbitrary system-dictated transaction aborts at all costs and give the designers greater control of the conflict handling process.

   (d) Negotiation modes, where clients can negotiate with each other for an agreeable solution when there are design conflicts.

3. Flexible and interactive lock management. Locking mechanisms must be interactive and flexible enough to allow multiple transactions to proceed concurrently without having to wait indefinitely for other transactions to complete.
In collaborative work, locks must be *shareable*. For example, instead of locking an object in the write mode (making it inaccessible to others), a designer may *share* the write lock with another designer in a group, so that they can both modify the object and be notified whenever that happens. Lock communication modes would keep users aware of the lock status of an object, so that they may do other useful work while waiting for locks on objects. Finally, the two-phase locking protocol is probably too restrictive for collaborative work. Users must have the flexibility to acquire and release locks *anytime* during a transaction.

4. **Client, data and transaction records.** A collaborative design framework needs to maintain extensive records of the clients using the system and their activities (transactions), the data used, dependencies between clients and data, and timestamping of data and operations. This record keeping is useful for establishing inter-client communication, locking and update notification, conflict detection and negotiation, and for keeping track of design evolution.

5. **Version management of data.** Version management is necessary for promoting concurrency of the design process, since different designers may work in parallel on their own versions and merge them at some later time. Object versioning also helps to keep track of the *evolution* of the design process.

6. **Capturing of design rationale.** A collaborative design environment must capture the rationale behind data changes and modifications by the designers, so that error detection, conflict handling and design rollback is facilitated, and the design process itself is well documented. The system must query the designers regarding why changes were made and record this information along with a timestamp.

7. **Flexible concurrency control.** Serializability is too restrictive for collaborative engineering since it enforces atomicity of transactions and does not allow interaction between users and their transactions in the middle of their execution. A collaborative transaction management framework must take into account the *semantics* of the transactions and the data being operated upon, so that the notion of “data correctness” may be more defined beyond the limited criteria set by an ordered sequence of reads and writes. This may be achieved using the concepts of transaction nesting and grouping, discussed next.

8. **Nesting and grouping of transactions.** A transaction should be divisible into smaller and nested subtransactions to any arbitrary depth. This allows a hierarchical subdivision of design effort and provides smaller units of control for transaction management functions such as commits, logging, rollback and recovery. It should also be possible for several small transactions of different users to be grouped together into one common transaction, so that they may actively share data between themselves instead of having to access data through transaction boundaries.
9. **Shared transaction processing.** The concept of grouped transactions necessitates the need for *shared* transactions by multiple users. Conventionally, clients interact with each other *across* individual transaction boundaries. This involves transaction initiations and commitments to a shared database. In circumstances where more active and quick data sharing is required, typically between a few designers working on a very specific area of design, or negotiation for a common design object, the designers should be able to participate in a *shared transaction*. A shared transaction is a *single* atomic transaction with multiple participants. Since all the designers share the same data pool (the data pool of the transaction), there is no need to cross transaction boundaries to access changes to data made by others. This enables quicker exchange of ideas and faster processing.

10. **Database partitioning and encapsulation.** The functional subdivision of tasks should lead to *structural partitioning* of the database, with each partition or area serving one or more divisions of design activity. Each designer must have a *private dataspaces* for conducting personal work not accessible to others (unless explicitly authorized to do so). Several engineers needing to work as group should have a shared database area accessible to all of them, and inaccessible to others outside the group. In this way, transient design changes may be *encapsulated* from non-cooperating users, who are not affected by inconsistent data states during the design process. As with nesting of transactions, database partitions may be nested, i.e., they may be sub-divided, and be assimilated into the parents.

11. **Exploitation of application semantics.** In localized shared databases insulated from non-cooperating users, consistency conditions on data can be relaxed considerably. Application semantics may be used to determine the validate the consistency of data rather than by the use of serializability.

12. **Robust conflict handling.** A collaborative engineering environment must support schemes for detecting and handling conflicts between clients and their database operations. In traditional database systems, the system aborts (often arbitrarily) one or more conflicting transactions in order to resolve the conflict, without informing the affected clients first. The proposed framework must attempt to avoid system dictated aborts at all costs. Rather, the conflict resolution process must be based on operation semantics, data constraints, and inter-client negotiation.

13. **Rigorous constraint management.** A collaborative CAD framework must support rigorous constraint management utilities for ensuring semantics-based data consistency. For example, the function which updates an attribute of an object must check to see whether the new value is valid and lies within acceptable bounds, and to inform the updater if it is otherwise.

14. **Graphical information representation.** Graphical representation of information is of paramount importance in collaborative work between several users.
This includes utilities like mailboxes, communication ports, dynamic version tree browsers, data structure browsers for classes and objects, hypertext facilities for navigation through the database, and multimedia negotiation facilities.

1.5 Thesis Outline

This chapter has introduced the primary objectives and scope of this study, which is to develop a transaction management environment for supporting collaborative engineering as proposed in the DICE framework. The remainder of the thesis is divided into four chapters.

- Chapter 2 provides a general background on various issues concerned with this study, such as concepts of object-oriented databases and OODBMS requirements for CAD, the advantages of using OODBMS over RDBMS for such work.

- Chapter 3 presents a detailed overview of ONTOS, the commercial OODBMS that was used for implementation purposes, and some of its transaction management deficiencies.

- Chapter 4 presents our model for a collaborative CAD environment, and discusses the advanced transaction management functionalities required to support the framework.

- Chapter 5 provides implementation details of various transaction management features using ONTOS.

- Finally, Chapter 6 concludes this study with a discussion of present implementation problems and future directions of research.
Chapter 2

Background

2.1 Introduction - OODBMS

The field of object-oriented database management systems (OODBMS) has emerged as a convergence of several research threads. The fields of object-oriented programming languages, artificial intelligence, software engineering, data abstraction theories and complex data management have all contributed to the use of object-oriented technology in the database area. The applied areas that have necessitated OODBMS capabilities include large computer-aided design and engineering (CAD/CAE) applications, distributed cooperative computing, computer-aided software engineering (CASE), and office information systems (OIS). The development of OODBMS have been facilitated by the availability of high performance graphics workstations and networking environments that support computer-aided design.

2.1.1 Characteristics of OODBMS

An OODBMS is a database management system which provides all the facilities of a conventional database system, namely,

1. A data model: it provides a set of templates or structures which are used to model the information, a set of operations to manipulate them, and a set of constraints to define the consistent state of the data;

2. Persistence: the data resides in persistence storage, rather than in volatile memory, and can be used across sessions;

3. Concurrency: multiple users access and use the same database simultaneously, and may interact with each other;

4. Transaction management: a process which monitors database interactions to ensure consistency or correctness and stability of the data;

5. Recovery: the ability to recover from a crash to some defined stable state;
6. **Query language**: a high-level, easy-to-use language for accessing information systematically;

7. **Performance**: efficient access structures and algorithms for retrieving large amounts of persistent data from secondary storage;

8. **Security**: protection of information from unauthorized access.

An OODBMS, however, is also an *object-oriented* system and supports the following additional capabilities [3, 19]:

1. **Data abstraction.** This allows the development and use of abstract and logically complex and flexible data types.

2. **Powerful information modeling capabilities.** Information is modeled in the form of classes and objects which *represent and capture the structure and behavior of real world entities* in the computer environment, making them ideal for simulation and design purposes.

3. **Object identity.** The system defines and maintains unique identifiers for objects. This allows “equal” objects (which have the same attributes and equal attribute values) to coexist; it frees the user from the need to define unique keys for entity instances;

4. **Encapsulation and data-hiding.** Objects are manipulated by operations that are defined on their types, and the innards are hidden (or sometimes inaccessible as in certain language implementations) from external access. The implementation of these operations may change without invalidating their use;

5. **Active (intelligent) data.** The encapsulation of procedures along with the data gives it the ability to *reason* about its domain, consistency, validity, etc.; this enhances the capability to define *triggers* on the database for intelligent operations.

6. **Inheritance.** This allows the reuse and/or incremental redefinition of new class structures in terms of existing ones, and enable the “passing-down” of information to derived objects. Two facets of inheritance are:

   (a) **Specialization**: entity structure and behavior may be successively specialized by *class derivation*; and

   (b) **Generalization**: Entities having common attributes and behavior (i.e., similar) may be *generalized* to inherit the common properties from a common superclass while dissimilar attributes may be specialized inside each of the child classes.

7. **Polymorphic data and functions.** This is a feature by means of which data dynamically assumes various forms determined at runtime.
8. **Composition.** Objects may be *part-of* another object; this is different from aggregation, which refers to collection objects such as sets and arrays.

9. **Message passing.** The interaction of objects by the invocation of each others’ methods - a feature that enhances simulation.

10. **Extensibility.** The set of operations, structures and constraints available to applications are not fixed as in traditional DBMS; users may define new operations and types, incrementally into the application.

In short, the flexibility and power of object data representation combined with DBMS facilities provides a powerful medium for modeling, coordinating, storing and manipulating engineering information.

We first discuss the various requirements of a database system for supporting collaborative engineering design, and show how OODBMS are eminently suitable for this purpose. Next, we consider the advantages of OODBMS over RDBMS for CAD applications.

### 2.2 Requirements of a Database System for Collaborative Engineering Applications

A large computer-aided-design or engineering (CAD/CAE) project typically involves a group of designers working cooperatively on distributed workstations to complete a complex design by closely interacting between themselves and dynamically sharing design data and information. Such design environments necessitate powerful data modeling, sharing and management tools, communication protocols, and a flexible transaction framework for concurrency management of highly interleaved and interactive transactions. The essential features of a data management system that are required for supporting such applications may be summarized as follows.

1. **Complex information modeling capabilities.** Engineering data representation is normally complex because of the complexity of the physical systems that have to be modeled and designed. Design entities are also intricately inter-related by various functional or structural links (such as *component-of* or *complement-of* relationships). Design of engineering systems also involve *simulation* whereby designers test behavior of prototypes before final design. This requires various simulation capabilities, such as representation of entity dependencies, message passing, scheduling, etc. The object-oriented data model allows the user to define data structure of any level of complexity with great flexibility. Message passing between objects makes object-oriented programming (OOP) ideal for simulation.

2. **Semantic schema design.** Because of the complexity of engineering data, large database schemas must reflect design semantics and hierarchy. Otherwise, maintenance and extension become problematic. This is particularly significant when complex inter-object relationships and dependencies have to be
represented in the schema. Inheritance hierarchies, object compositions and references, and polymorphism are some of the ways OOP facilitates designers to realistically represent complex design information in the database schema.

3. **Rigorous constraint management.** Due to the the size and complexity of engineering databases, consistency of data state must be maintained by enforcing rigorous design constraints as the data evolves. Consistency constraints may be of several forms - data types, valid ranges, design conditions, safety limits, etc. Often, the best way to enforce these constraints is to embed the constraint checking mechanisms in the data structure itself, rather than have extraneous programs check for consistency for every data update. Data in the form of objects allow a very convenient way to implement this scheme, since the class methods for updating each attribute of an object may also perform the constraint and consistency checking upon each update.

4. **Data management about data.** Since CAD applications are highly data-intensive and evolve incrementally, it is necessary for the DBMS to maintain elaborate data records about the design data itself. This includes information such as ownership, time and purpose of creation, update histories, client dependencies (i.e., which clients have accessed the data), lock status, versions and several other advanced issues related to transaction management. Again, it would be natural to encapsulate these pieces of information about design data inside the design objects themselves, so that update and maintenance of records is automated and the task is distributed between the various objects rather than being controlled by external mechanisms.

5. **Data sharing.** One of the key issues in collaborative engineering is the sharing of data between various designers. It should be possible to partition or group data based on various criteria, such as ownership, use, time and purpose of creation, inter-data relationships, distribution over a network, or any other meaningful purpose. Data used by clients must also be capable of interaction. The object-oriented data model facilitates object sharing between clients because the discriminating criteria is encapsulated within the objects themselves. Message passing enables design objects to interact as the entities they represent do in the real world.

6. **Data versioning.** Data must also be versionable so that different versions co-exist in the database and data updates do not result in the overwriting of old data. Objects provide a convenient unit for efficient storage and version management of data. Versioning of data significantly promotes concurrency (since different designers may work concurrently on different versions of the same object instead of having to wait for each other release it). It also helps to keep track of the evolution of design, since objects may store their version history. In the event that a design appears to be faulty at any stage, it should be possible to rollback the design to some valid data state.
7. **Inter-client communication.** One of the most important requirements of an CAD database environment is effective *communication* protocols between designers, a feature that is conspicuously absent in present DBMS environments. Designers are often unaware of each others' developments, and this may lead to lack of coordination, reduced parallelism, a considerable waste of time and resources, and sometimes even faulty designs due to misinterpretations of data. Inter-client communication facilitates active exchange of information, updates on data modifications or data status, better synchronization of work across design interfaces, and greater concurrency, all of which are essential for collaborative development. Various communication modes may include

(a) *Lock communication modes:* these inform the users of the lock status of objects, so that they may schedule other useful work without waiting in ignorance if a requested object happens to be locked by another client;

(b) *Update communication modes:* these notify appropriate users of object changes and updates;

(c) *Conflict communication modes:* these notify users in the event of a conflict between two or more transactions (for example, a deadlock or commit failure); and

(d) *Negotiation modes:* a more advanced communication protocol for negotiation between clients for a mutually agreeable design.

8. **Flexible transaction framework.** Collaborative engineering is based on the notion that *units of work must interact* so that the results are usable together. This means that concurrent transactions must be able to communicate their results to each other in the middle of their execution. Since CAD transactions are normally of long duration, locking mechanisms must be flexible enough to allow greater concurrency, so that transactions do not have to wait indefinitely for each other to complete. In addition, facilities must exist for sub-transactions to be *nested* within another (representing sub-division of design tasks), and to be *grouped* for the purposes of active data sharing. All these advanced schemes violate the traditional notion of *serializability*, which preserves database consistency irrespective of the nature of the transactions; however, it significantly reduces concurrency, and is largely unsuitable for CAD environments. Data "correctness" therefore will have to be ensured by other means, which take into account the *semantics* of the transactions and data types. Object-oriented databases present the opportunity to to provide greater concurrency using this scheme [45], because database operations on types reason about their domain, and have more semantic content than just read or write. A detailed discussion of advanced transaction management features for collaborative engineering will be presented in Chapter 4.

9. **Efficient storage mechanisms for fast data access and retrieval.** Interactive graphics-based CAD transactions necessitate fast retrieval and access
to persistent data. Related data may need to be \textit{contiguously clustered} on secondary storage according to various criteria (for example, all the components of a machine part, or all versions of the same data, etc.). Many CAD applications involve the use of graphics and multimedia information, which require special data compaction, storage, mapping and access mechanisms. Associate retrieval (based on object identity and inter-object references) is another feature that may help speed up queries in object-based databases. Distributed computation may also necessitate \textit{distribution of data} on a network across heterogeneous platforms.

10. \textbf{Computationally complete database programming language.} Most engineering applications involve complex mathematical manipulations of data or systems level programming that necessitate a computationally complete or sufficiently low level database definition and programming language. It would be difficult, for instance, to perform engineering design using SQL.

11. \textbf{Compatibility, extensibility and integration.} OODBMS for engineering are meant to be used as a tool integrated in a bigger and diverse CAD environment; hence it is very important for these components to interface to each other easily. Having common data representation or data manipulation language is one of the ways to achieve compatibility. The OODBMS system should also be customizable to suit the needs of the application developer.

12. \textbf{Graphical development environment.} Though not a key requirement, support for a graphical environment is necessary for most CAD applications. This is useful for browsing and modifying data structures and dependencies, representing engineering drawings and charts, and general ease of application development.

\section*{2.3 Advantages of OODBMS over RDBMS for Engineering Applications}

The design of traditional relational database systems has largely been determined in response to the needs of typical business applications. These systems are well equipped to store and manipulate flat data that can be tabulated (and therefore have a fixed structure), accessed and modified by very high level query languages, and presented in attractive forms. However, the set of structures, operations, and constraints in relational data is limited and fixed. Consequently, any structure and operation needed by the applications should be mapped into this limited set. As the applications become more complex, this mapping becomes onerous and unrealistic.

Computer-aided design applications are highly data intensive, and involve complex data representation to emulate the structure and behavior of complicated entities that have to be designed. The relationships between entities is often too complex or subtle to be modeled by the relational model (for example the part-of relationship). Besides, a considerable amount of reasoning intelligence is associated with the design data
in the form of feasibility, consistency, and other constraints imposed by the design process. These constraints may be too cumbersome to implement using the system of constraints used in the relational systems. In other words, engineering information is not often simple enough to be represented in the form of flat "dumb" relations, and the associated constraints in a RDBMS. Thus, the complexity of engineering data necessitates appropriate level of extensibility to capture application-specific data semantics and mechanisms for incremental development of database structures for which the relational system is a poor tool.

We briefly enumerate the advantages of OODBMS over RDBMS.

1. **A more realistic data model.** Design classes and objects in an OODBMS represent real-world design entities, and gives a much better feel for the *mechanics* of the problem than do a set of flat tables as in a RDBMS. Also, there is no mechanism in the relational model to associate behavior (functions) with data, which is achieved easily with objects.

2. **A more powerful data model.** Object data representation is highly *flexible,* and may be customized by the users with little restriction. It allows data to be *polymorphic* and *intelligent.* Facilities for *inheritance* and *schema evolution* allow the design to grow incrementally. Problems which normally occur in a RDBMS, such as normalization and unnecessary duplication of data do not exist. The complexity of data and intricacies of relationships that can be handled by the object model is far superior to that of the relational.

3. **Easier schema development.** In OODBMS, generalization and inheritance allow the schema to be better structured, more intuitive, and to capture the semantics of the application. The schema is also smaller because of inheritance, since common attributes may be factored out (generalization). Since design environments are characterized by continual change, the type and schema definitions are likely to be modified as designers arrive at better understanding of their problem. Conventional RDBMS systems have only limited facilities to accommodate changes at the level of types. OODBMS are more flexible and some systems (e.g., ORION) provide extensive schema evolution facilities.

4. **Impedance Mismatch in RDBMS.** One of the problems in developing complex applications using a RDBMS is the *impedance mismatch* between the database manipulation language (DML) and the general-purpose programming language in which the rest of the application is written [4]. There are two aspects of this mismatch: (a) difference in programming paradigms, for example between a declarative language such as SQL and an imperative programming language such as C; and (b) a mismatch in the type systems, whereby loss of information occurs at the interface. In most relational systems, the DML *lacks computational completeness* to express complex mathematical manipulations of data common to engineering design. Most OODBMS on the other hand provide database extensions to *computationally complete* programming languages (e.g., C++, CLOS, Smalltalk) which may be capable of handling complexities typical of large scale programs such as finite element analysis.
5. **Object identity.** The relational database model is *value-based*, which expresses relationships between two objects by embedding the same (or equal) value in two or more related objects. The user often has to define unique keys to distinguish between similar entities, and this often results in duplication of data. Object-oriented systems are *identity based*, and can relate two or more objects independently of their embedded values (value-based frameworks can also be incorporated in OODBMS if needed). Identity facilitates the notion of *sharing* by having references from various points to the same object, rather than keeping copies everywhere.

6. ** Powerful unified knowledge representation.** Object-oriented representation of information provides an uniform framework for integrating *data* (attributes) and *knowledge* (procedures). Several powerful artificial intelligence techniques appear in OODBMS. Examples include classification and specialization in hierarchies (inheritance captures relationships elegantly and allows propagation of knowledge), delegation of behavior, polymorphism and run-time binding of messages to methods, automatic garbage collection and memory management strategies, and intelligent behavior of entities. Most engineering databases are intended for use as a *tool integrated in a bigger CAD environment* rather than as a stand alone, for example with graphics packages and expert systems. The use of objects as a data structure *fuzzies the distinction between “knowledge” and “data”* and necessitates no separate representation, since the entity-domain knowledge is embedded in the object itself. A RDBMS provides no comparable facility. OODBMS are better equipped to play a key role in the development of next generation integrated tools such as *expert or intelligent databases* [30]. One such system is COSMOS [41], under development at IESL, MIT.

7. **Better transaction management and concurrency.** Current RD-BMS enforce *serializability* of concurrent transactions in order to maintain database consistency. OODBMS present an opportunity to provide greater concurrency than more traditional approaches allow [45]. This is because in the object-oriented approach, the database systems knows more about the operations that are being performed - they are not simply read or writes, but rather have greater semantics. “Correctness” can therefore be enforced on the basis of application semantics rather than using the no-read-write conflict approach. This is discussed in greater detail in Chapter 4.

8. **Better version management of data.** OODBMS allow version management of data at various levels of granularity, such as at the level of attributes and methods, objects, classes and even database hierarchy. RDBMS allow version management only at the granularity of entire tables, and not at the level of tuples and attributes. Thus version management in RDBMS is expensive. OODBMS provide better frameworks for maintaining versions of *configurations* such as composite objects, collections, etc. OODBMS also allow *intelligent version control* facilities by having methods in the objects which check for dif-
ferences (also called “deltas”) between versions of objects to decide whether new versions should be created or existing versions should be merged. An OODBMS thus helps to keep better track of the design evolution in versions of the data, and enables easy rollback to a stable data state if the design is found to be faulty at any stage.

9. More expressive query language. Most RDBMS support excellent standardized high-level query languages. Current OODBMS lack a high level standard query language. However, various object-oriented query formalisms are currently under development [34]. Object-SQL (or OSQL) is an object-oriented approach to SQL. OSQL allows function invocations in query statements and associative retrieval through object references. This allows a single query to do much more (for example by performing complex calculations by calling functions) than a standard SQL query which can perform only a limited set of complex operations. Object identifiers and functional composition of queries may also eliminate expensive joins that may be required in SQL queries. In addition, accessing data through inter-object references (popularly called “pointer-chasing”) is much more efficient than navigation through relational links.

10. Better support for cooperative work. OODBMS are better candidates for supporting environments for collaborative work than RDBMS. OODBMS allow more complex data handling, support sharing of artifacts at multiple levels of granularity, support versions and alternatives better, and can capture the mass of management information that goes along with any multiperson project - such as schedules, task dependencies, annotations, records, design histories, design decisions, and even project-specific policies.

2.4 Summary

In this chapter we have presented the various characteristics of OODBMS and emphasized their amenability for engineering applications. We have also discussed their advantages over traditional RDBMS. Case studies of a comparative analysis of the effectiveness of RDBMS and OODBMS have been conducted [19] for a simple computer-integrated manufacturing system. The results indicate that the object-oriented prototype has a superior schema, is capable of providing convenient access to information, and is easier to extend and maintain.

The next chapter presents an overview of a typical commercial OODBMS, ON-TOS, which was used in our implementation, and critically examines its features in the light of the requirements stated above.
Chapter 3

The ONTOS Commercial OODBMS

3.1 Introduction

Object-oriented programming is gradually evolving into a popular and standard paradigm for large software application development. Typical examples include computer-aided design and engineering (CAD/CAE), computer-aided software engineering (CASE), and office information systems (OIS). For such applications that are data or knowledge intensive, and require collaborative participation, traditional storage mechanisms, such as application specific file structures are grossly inadequate, and necessitate a database system for information management. Object-oriented databases provide persistence to design objects, concurrency, consistency and all the other features of a database management system. The object-oriented programming paradigm combined with DBMS facilities provides a powerful medium for modeling, coordination, storage and manipulation of engineering information.

Several OODBMS have been developed recently. These include ORION, ITASCA, GEMSTONE, ONTOS, Versant, ObjectStore and ObServex/ENCORE. A detailed comparison of various features of these systems can be found in [2]. In this chapter, we discuss and critically review the principal features of the ONTOS commercial OODBMS, which has been used for the implementation of our transaction management framework, in terms of the facilities (or deficiencies) it offers for cooperative engineering product development. It will also serve as a general overview of the functionalities of OODBMS in general.

ONTOS [53], formally VBase, is a commercial OODBMS developed by Ontologic, Inc., MA. ONTOS uses C++ as the front-end object data definition and manipulation language (DDML), provides persistence to classes and objects created in C++ (version 2.0), and extends some of its features. An interpretive fourth generation language (4GL) is currently under development. ONTOS also interfaces with PTech, a graphical tool for database schema generation. Version 1.6 of ONTOS is currently available. It runs on Sun3, Sun4, DEC and Apollo workstations and in UNIX/ULTRIX, VAX/VMS and OS2 environments.

The various features of ONTOS discussed include: system architecture, advanced
data modeling, dynamic schema evolution, storage management, transaction management features such as locking and concurrency management, conflict handling, facilities for collaborative engineering applications, query management and user interfaces.

3.2 ONTOS System Architecture

ONTOS uses a distributed client-server architecture as shown in Figure 3-1. In the present release, all the server machines (or nodes) must be homogeneous (i.e., belonging to the same hardware family and running the same operating system). The database is physically distributed over the server machines on the network, but provides an uniform object reference space to the application (i.e., the application on any client machine does not have to know where the data actually resides). The client is created by linking the ONTOS Client Library into the user's application. The server runs on every node containing a portion of the database and fulfills data requests by clients running on behalf of the application programs. The server also controls storage management, such as data movement to and from the disks, clustering, etc., and functionalities of transaction management. The Client Library in a library of C++ classes and functions. It provides classes for persistence, schema definition, modeling of multi-valued attributes (such as aggregates and collections), etc.

3.3 Basic Data Modeling Using ONTOS

ONTOS uses C++ 2.0 [27] as the DDML, and therefore provides a hybrid object-oriented programming system. Classes and functions are defined in standard C++ code which makes ONTOS classes portable, and existing C++ code (from other applications) may be easily incorporated into ONTOS. C++ classes provide support for static data (seen by all instances of a class) and static function members (which may be invoked directly from a class without instantiating it). In addition, C++ allows true encapsulation by data hiding (in the form of private and protected variables). C++ supports multiple inheritance, but does provide for alternative name conflict strategies. In the event of a name conflict, the earlier base class overrides the others. However, ONTOS version 1.6 does not support multiply inherited persistent classes and objects.

Unlike other object-oriented programming languages such as Smalltalk and LOOPS, C++ is a static programming language and does not provide a flexible run-time environment, which makes schema evolution difficult (schema evolution is discussed shortly). It is also a strongly typed language. ONTOS provides a library of C++ utility classes for application development. It also provides exception handling mechanisms for trapping exceptions raised during the execution of the application.
Figure 3-1: ONTOS distributed client-server architecture
3.4 Composite Object Facilities

In engineering design, design entities are often composed or made up of several component entities or parts. For example, a car is made up of several parts, which themselves have sub-parts (Figure 3-2). In the object-oriented paradigm, such entities may be effectively modeled as composite objects where a set of objects belongs to (or is-part-of) another object which may be treated as a single logical entity. A composite object is an object with a hierarchy of exclusive component objects, called composite object hierarchy. A discussion on the semantics of the structure and behavior of composite object can be found in [6, 20].

Neither C++ nor ONTOS supports part-of relationships between objects. However, a class may have an instance of another class as one of its member attributes. This essentially represents a pointer (or “weak”) reference from one object to another. A composite object model for C++ is presently under development at IESL [41, 49].

3.5 Dynamic Schema Evolution Facilities

Engineering design applications require considerable flexibility in dynamically defining and modifying database schema, i.e. class definitions, inheritance structure and specifications of attributes and methods without requiring application shutdown. This is necessary because design is an incremental process and evolves with time. Schema modification is intimately tied with the issue of version management, which will be discussed shortly.

3.5.1 Taxonomy of Schema Evolution

Typical changes to the schema include the following [6]:

1. Changes to the contents of a node (class or its instances);
2. Changes to an edge (i.e. the relationships between classes); and
3. Changes to a node in a class lattice.

The various types of changes are explained and exemplified in the following sections.

Changes to the contents of a node

1. Changes to instance and class variables. An instance variable may be inherited and modified by any instance, while the value of a class variable can be inherited but cannot be changed by the instances or the subclasses. The various types of changes that the designer may impose on classes and instances are:
Figure 3.2: Composite objects in a Car
(a) *Add a new instance variable.* A new slot may be added to hold additional design information. For example, the number of beams supporting a floor slab may be added to as an instance variable to the class *Floor.*

(b) *Drop an instance variable.* At an advanced stage of the design, some information may become redundant or unnecessary. Hence, the slot holding the information may be deleted to reduce the size of the design object. For example, if a class containing a slot $z$ is made a subclass or component class of another class having a slot containing the same data, then information may be retrieved through inheritance rather than having storage allocated in all individual instances. Thus, slot $z$ may be dropped.

(c) *Change the name of an instance variable.* Often it may be necessary to identify an instance variable by another name; there may be a name conflict with an inherited variable, or just for the ease of future reference. For example, the user may want to change the *Weight_in_pounds* slot of a *Beam* object to *Weight.*

(d) *Change the domain of a variable.* There may be a need to change the type of information (e.g., integer, real, characters, etc.) a variable (slot) may store.

(e) *Change the default value of an instance variable.* This may be necessary if design specifications need to be altered at some stage. For example, after the design of all concrete beams are complete, one may set the default value of the variable *material-type* of all *Beam* objects to *steel,* so that the designer does not have to set the value for every object s/he instantiates later.

(f) *Change the inheritance specifications of an instance variable.* This may be necessary if the designer wants to override the information derived for a slot, and specify his/her own specifications. Also, in the case of multiple inheritance, where more than one parent has slots of the same name, the user may choose to inherit the slot from the desired superclass (supertype). For example, assume that *Reinforced_Concrete_Beam* and *Prestressed_Concrete_Beam* are superclasses (parents) of *Beam* and the inheritance specification for the default value of the Beam’s *material* slot is set to *Reinforced_Concrete_Beam.* At a later stage this inheritance specification could be changed from *Reinforced_Concrete_Beam* to *Prestressed_Concrete_Beam,* so that *material* would inherit the default value from *Prestressed_Concrete_Beam.*

(g) *Add a class variable.* This allows addition of new variables that may not have been perceived when the classes where designed. For example, a new variable (slot) *reinforcement_ratio* could be added to a *Beam* object.

(h) *Drop a class variable.* A class variable may be dropped when it is no longer considered necessary for the purposes of design.

(i) *Change the default value of a class variable.* This is necessary if the design specifications have changed at some later stage, and all instances of a class
need to be informed about the change.

2. Changes to methods. These include:

(a) **Add a new method to a class**, to allow incremental design development; procedures for analysis or design of various sub-components may be added at any time. For example, there may be methods that perform a preliminary cost estimate (especially during preliminary design). Later, methods that perform a detailed cost estimate may be available. These methods can be added to the appropriate design objects.

(b) **Drop an existing method**, when it is no longer useful.

(c) **Change the name of a method**, for the ease of identification or if its functionality has changed.

(d) **Change the source code of a method**, which may be done if there is an error, or if the method is reimplemented to perform some other task.

(e) **Change the inheritance of a method**, which may be done under circumstances similar to that for a variable.

Changes to an edge

These modifications concern changing the hierarchy of classes in the inheritance lattice. They include:

1. **Make a class a superclass of another class**. This is done to abstract the common attributes of several classes into one super class, or to allow the new subclass to inherit several additional attributes from the new superclass. For example, in order to lend graphic abilities to all instances of the class **Beam**, **Beam** could be made a subclass of **Geometric_Obj**, which contains all the procedures for graphic display of objects.

2. **Remove one of the superclasses of a class**. This may be done when certain inherited slots are not perceived necessary, or are in type-conflict with attributes inherited from another superclass.

3. **Change the order of superclasses of a class**. This involves reshuffling the order of the class hierarchy, and is done to reflect major changes in design specifications.

Changes to a class

The following changes to a class type need to be supported:

1. **Add a new class**. This is done to introduce a class of new design objects at any stage of the design. For example, a new class **T_Beam** could be added as a subclass of a **Beam** object.
2. **Dropping an existing class.** A class may be deleted when it is no longer necessary.

3. **Changing the name of a class.** This may be done in order to avoid name-conflicts or for ease of identification.

### 3.5.2 Schema Evolution in ONTOS

ONTOS is quite limited in its schema evolution facilities largely due to the static nature of the C++ language. New classes (subclasses of existing classes) may be *programmatically* defined at run time, and the C++ code for the class may be automatically generated. The structure of a class may be modified at run-time if it has no instances (data migration is not supported). ONTOS comes with a graphical schema designer for schema evolution and modification. A *dynamic linker* for C++ has been developed (described in Chapter 5) at IESL which allows some dynamic schema evolution facilities.

### 3.6 Object Storage Management

The storage subsystem of an OODBMS is responsible for management of secondary storage of objects on the disk, and in-core memory management in the client machines. It includes allocation and deallocation of pages on disk, movement of pages to and from the disk and the client machine memory, object clustering and indexing on collections, object buffering in memory, etc.

As discussed earlier, ONTOS supports a homogeneous distributed client-server architecture for storage management. ONTOS objects may either be persistent (if derived from system class *Object* or subclasses thereof) or non-persistent (no disk representation). Objects are uniquely identified by unique identifiers (UIDs). ONTOS provides two kinds of memory-based references - *transparent* (called *TRef*), and *direct* references. Direct references require an explicit *activation* call to load the object into machine memory. Transparent references are pointers to instances of kernel class *TRef*; objects pointed to by *TRefs* are automatically activated upon reference. Object *deactivation* flushes an object to the current transaction buffer pool and are written to the database when the transaction commits. Activation and deactivation are performed using a set of *getObject()* and *putObject()* functions (Figure 3-3).

ONTOS allows the user to do his/her own memory allocation once the objects are activated. For example, the user may want to co-locate certain objects with others within the process heap to minimize virtual memory paging. The user then allocates a block of memory using memory allocation functions (C++ provides functions for memory allocation), and supplies the block’s location and size to the activation function, whereby objects are allocated into the supplied block. Objects may be activated or deactivated singly or in clusters (for example, a whole aggregate object), see Figure 3-4.

ONTOS provides three *object buffering* schemes for data transfer to the disk: (a) no-buffering, where each “put” call results in immediate transfer to the server; (b)
Figure 3-3: Object activation and deactivation
Figure 3-4: Granularity of activation and deactivation functions
default-buffering, where objects are buffered and sent in small groups to the server, normally once after every 10 “put” calls; and (c) buffer-until-commit, where objects are buffered in the transaction buffer pool until the transaction is committed.

ONTOS allows clustering of objects on the disk according to the specifications of the user. Objects may be clustered according to classification (instances of the same class), aggregation (members of a collection, e.g., a Set), reference (e.g., object A references object B) or association (for instance, several objects share the same value of a given attribute, all Car objects having blue color, etc.).

ONTOS does not allow database partitioning (discussed in Chapter 4) into local databases shared between specific users. Also, it allows only one database to be opened and accessed in the whole application.

3.7 Basic Transaction Management Features

In this section, we discuss some of the basic features of the transaction management framework of ONTOS. The various components of this framework, such as lock management, version and notification management, concurrency management, etc., are discussed in subsequent sections.

ONTOS transactions are defined statically at compile time in C++ code. ONTOS provides a set of exceptions to handle access conflicts, and possibilities of transaction wait, abort or deadlock. These exceptions enable the application to deal with the problem appropriately, or to do something else while waiting for another transaction to complete. ONTOS does not support nested and grouped transactions (these are discussed in detail in Chapter 4). An user may initiate only one transaction at a given time. Also, ONTOS transactions may access only one database at a time. A future version of ONTOS (Release 2.0) is slated to support nested and grouped transactions.

3.8 Client/Object/Transaction Data Management

ONTOS does not export to the application any information regarding the users, and their activities, object ownership, timestamp and usage, lock status of objects, transaction ids and their status, etc.

3.9 Lock Management

ONTOS has four lock types: read, write, writeintent and default. The strength of the last two locks are determined by the system according to the tolerance specified by the transaction. There are two tolerance levels: (a) read-write conflict, which prevents a readlock from being granted on an object that has been granted a writeintent lock, and (b) no-read-write conflict which allows a readlock on a writeintent locked object and attempts to resolve access conflicts at transaction commit time.

ONTOS enforces two-phase locking with serializability for concurrency control. Therefore it does not allow locks to be released at will during a transaction. All locks
are released at the end of the transaction, when it either commits or aborts.

ONTOS locks may be programmed to be set at various levels of granularity, viz, (a) individual objects, (b) references to objects, (c) collections and aggregates, (d) extension (i.e. all instances) of a class, etc. ONTOS does not support locking of class structure or class lattice locking.

ONTOS does not support inter-client communication, but queues lock requests and raises selected exceptions in the event of lock or access conflicts. If the exceptions are not trapped, it aborts the conflicting transactions without notice. It also does not provide the application the lock status of an object at any time, nor does it notify the client about the status of lock request. These enhancements have been built over ONTOS (see Chapter 5).

3.10 Communication and Change Notification

Inter-client communication is of paramount importance in distributed cooperative engineering work such as CAD/CAM, since designers share huge amounts of data, and hence need to be informed of its status. ONTOS does not support any form of inter-client communication or update notification schemes. At IESL, these features have been implemented over ONTOS as discussed in Chapter 5.

3.11 Version Management

Version management is important for engineering databases because design is often an experimental and incremental process, the scope of which changes with time. It is necessary to keep track of the evolution of design objects, and the changes made to a design by various transactions. It is also important for concurrent cooperative work, since different clients may work simultaneously on different versions of the same object, rather than wait for each others’ transactions to complete. Version management for collaborative engineering is discussed in detail in Chapter 4.

ONTOS does not support any form of version management. A version management scheme has been developed in ONTOS, and its implementation is discussed in Chapter 5.

3.12 Concurrency Management

Concurrency control is necessary in order to maintain database consistency and integrity. Traditional DBMS systems tend to enforce serializability of atomic transactions, and allow database updates in units of a whole transaction - either all the objects in the transaction are committed or none at all. Although this may work well for short-duration financial or business type transactions, it may often be too restrictive for concurrent engineering or CAD applications, where transactions are typically complex, unusually long, and are interactive and interleaved. An OODBMS
needs to provide a more flexible concurrency control environment than that dictated by serializability.

ONTOS attempts to serialize all concurrent transactions and enforces two-phase locking for maintenance of data consistency. It provides both optimistic and pessimistic concurrency control schemes according to the level of tolerance (no-read-write-conflict or read-write-conflict) set for the transactions. In a pessimistic situation, access conflicts are checked at the time a transaction is initiated; this reduces concurrency but ensures transaction commit when it eventually finishes. In optimistic concurrency control, access conflicts are checked at the time of transaction commit. This actually increases concurrency by pre-supposing that there will be no conflicts. However, since conflicting transactions are aborted arbitrarily by ONTOS, there is a risk of losing work done in a transaction.

ONTOS allows checkpointing of transactions before commit. Checkpointing commits all the objects in the transaction buffer pool immediately to the database without releasing the locks. This may promote data sharing.

3.13 Conflict Handling

ONTOS does not handle conflicts between database clients and their operations (such as lock or commit requests) well by default. It raises a set of exceptions when these conflicts are detected. If these are not trapped by the user’s application, ONTOS arbitrarily aborts one or more conflicting transaction, without any notification to the clients involved. It also does not support any mechanism to redo or undo a transaction.

3.14 Facilities for Cooperative Engineering

Traditional database environments enforce serializability of transactions, where one transaction is completely isolated from the effects of other concurrent transactions. Cooperative design (such as distributed complex CAD applications) however, is based on the notion that units of work must interact so that the results are usable together. This requires a very flexible concurrency control scheme, and the notion of database consistency and “correctness” to be defined by the requirements of the application.

ONTOS is not very suitable for collaborative application development, and offers almost none of the advanced features discussed above. Another disadvantage is that transactions are statically defined, hence they may not be changed at run-time (depending on how the design proceeds); and the static compile and run environment of C++ makes redesign and experimentation difficult and time consuming. It enforces serializability of transactions with two-phase locking. It does not support any communication or default conflict handling facilities. It does not support nested and grouped transactions, thereby precluding selective data sharing and interaction. It does not support the public and private database partitions. ONTOS lacks any form of database security (such as password protection or authorizations) which is also a hazard. Also, the lack of any form of records on users, objects and transactions make ONTOS ideal only for a single-user environment.

52
3.15 Query Management

Currently, most OODBMS lack an uniform, simple and efficient query language such as SQL in RDBMS. Object-oriented extensions of SQL (called object-SQL) have been developed for some systems, while others use their own query format.

ONTOS provides a programmatic SQL interface to its object base. It allows some of the basic SQL commands such as select, from, where, and, or and standard boolean and relational operators such as between, not between, is in, is not in, etc. One advantage of ONTOS object SQL is that procedures (or member functions) may be called within a query statement, to yield one or more of the parameters of the query. In addition, direct references in objects may be chained to any depth in query expressions, for e.g., it is legal to use an attribute in the following form: car.body.door.color. Associative retrieval in object SQL makes it more powerful and expressive than in the relational model.

ONTOS queries are executed by a library class, QueryIterator, which takes the SQL expression as a string argument and evaluates it. Queries are conducted against extensions of classes (the set of its instances) and aggregate objects. All queries may alternatively be implemented by the user in C++ using low level access and iterator functions provided by ONTOS. It also allows keyed access to data elements in dictionary and array aggregate objects.

3.16 Database Authorization and Security

Database security measures are necessary to guard against unauthorized access and tampering of data. Data authorization mechanisms help to control selective access to data (by privileged clients only) and to keep track of object ownership and usage.

Currently ONTOS provides no database security. Anybody with read access to the database file may read data, and with write access they can update it freely.

3.17 Multimedia Data Management

ONTOS currently does not support multimedia applications, though bit-mapped images (such as CAD drawings) may be stored as objects in the database. A multimedia presentation manager, STUDIO, is presently being developed.

3.18 User Interface

ONTOS provides a graphical schema designer (called XBrowse) developed using the X-Motif toolkit. It allows browsing of class hierarchies, object editing, dynamic creation of new classes, C++ code generation, and some database transaction facilities such as commit and rollback.
3.19 Summary

We have presented a comprehensive description of the principal functionalities of ONTOS, and its strengths and weaknesses for engineering design database management. It is a relatively new commercial product, and relies heavily on the power and portability of C++ as the programming language. It is clear that C++ is going to be the most popular object-oriented programming language, and several new products (such as Versant and ObjectStore) now provide database capabilities for C++ objects. In spite of the efficiency of C++, ONTOS suffers from the static nature and inflexibility of the language; and most advanced database management features are still under development. Some transaction management features have been implemented over ONTOS and are discussed in Chapter 5.
Chapter 4

A Transaction Management Framework for Collaborative CAD

4.1 Introduction

Efficient transaction management is one of the most important and yet one of the unresolved issues in collaborative engineering. This is because the nature of CAD transactions is very different from that of typical financial or business applications, and requirements also vary with the nature of the application. A typical CAD environment involves a group of designers working on a long and complex design by closely interacting among themselves and dynamically sharing design data and data about design data. Due to the diverse and complex needs of various types of CAD applications, no generic framework for transaction management for all types of applications has been suggested and none implemented so far. This chapter discusses a framework which addresses the collaborative CAD requirements of the DICE environment, and may be extended to other similar CAD applications as well.

The principal requirements of a transaction management framework for collaborative CAD have been discussed in detail Chapter 1. They are summarized here as follows:

- Data management about client, data and transactions
- Effective communication protocol between clients
- An interactive and interpretive transaction execution environment
- Shared and communicative object locking schemes
- Version management of data and design configurations, version tracking and design evolution
- A framework for design task discretization in the form of nested transactions
- Shared effort in the form of grouped transactions and partitioned database areas (private, group and public) to encapsulate transient design changes from non-cooperating users
- Relaxed concurrency control schemes which take into account the semantics of the transactions and data
- Conflict detection and handling protocols
- Negotiation framework
- Rigorous constraint management by embedding validity constraints on design data in the database
- Graphical information representation such as mailboxes, version browsers, etc.

In this chapter, we first enumerate the various functional modules of a transaction management system in a database management system. Then, we discuss briefly the features of transaction management in traditional databases and stress their limitations for collaborative CAD. Next, we present a model for collaborative CAD environment and discuss the functionalities of each of these transaction modules for supporting a such development environment. We will also continue to illustrate concepts presented in this chapter using the building design example presented in Chapter 1. We also provide a set of high level prototype operators or functions that accomplish these functionalities. Implementation issues are discussed in Chapter 5, and Appendix A provides the C++ code for various transaction management classes, their structure (member attributes) and behavior (member functions).

4.2 Transaction Management Functionalities in a DBMS

The transaction management system is responsible for maintaining database integrity while allowing execution of multiple concurrent transactions by various clients. The primary functional modules of a transaction management system in database management systems include (Figure 4-1):

1. **Transaction scheduling**, which is responsible for initiating, queuing, executing, terminating or aborting transactions;

2. **Concurrency control**, which ensures that concurrently executing transactions maintain database consistency;

3. **Locking facilities**, which controls the locking of objects, classes, etc.;

4. **Deadlock management**, which detects and resolves deadlock between transactions;

5. **Communication and update-notification facilities**, for client-server and inter-client correspondence;
Figure 4-1: Transaction management modules in a DBMS
6. **Version management**, which keeps a record of object changes and design evolution. Although version management is a complex sub-system by itself (independent of transaction management), versioning of objects plays a significant role in enabling greater concurrency, and will be briefly discussed in this perspective;

7. **Recovery management**, which provides facilities for database restoration from soft and hard system crashes.

We will first discuss how these transaction management features have been implemented in traditional database management systems and their limitations for CAD work. Subsequently, we will present enhanced functionalities required for supporting collaborative work.

### 4.3 A Brief Review of Traditional Transaction Management Frameworks and their Limitations for CAD

As a background to our study of collaborative transaction frameworks, we will briefly discuss transaction management functionalities in traditional database systems and their limitations for supporting collaborative work.

The traditional examples of database applications are banking systems, personnel files and airline reservation systems. These applications consist of entities which can be easily represented by data stored in tuples or records and logical groups of entities which can be represented in files or relations. Typical transactions in these applications are short and interactions of the transactions are well defined.

The salient transaction management features provided for such systems are briefly discussed below.

1. **Serializability.** The correctness criteria used for such applications is serializability and *two phase locking* is frequently used as the concurrency control protocol. A schedule of $n$ transactions is said to be serializable if it is equivalent (i.e. produce the same final result) to some *serial schedule* of these $n$ transactions. Serializability thus ensures database transition (due to concurrent transactions) from one state to another in a consistent manner irrespective of the nature of the participating transactions provided that they are all serializable. Serializability is founded on the assumption that individual concurrent transactions run oblivious to each other and do not interact in the middle of their execution. This is however unsuitable for CAD transactions. Collaborative engineering is based on the notion that *units of work must interact*, so that the results are *usable together*.

Serializability also assumes *atomicity* of transactions, which means that transactions either post *all* changes made during the transaction or *none at all*. In a CAD framework, this precludes experimentation with data and posting of the
correct or acceptable results only. Designers need the flexibility to selectively commit objects to the database at any time during the transaction (upon request from another user, for instance), and not the whole object buffer pool of the transaction.

Two-phase locking in traditional databases implies two phases of a transaction: (a) a growing phase, when locks on objects used during the transaction may be acquired, but none can be released; and (b) a shrinking phase during which locks may be released, but no new locks can be acquired. Two-phase locking guarantees serializability of transactions. However, this means that one cannot release locks on data (that one is done with) anytime during a transaction; in addition locks on data that will be used only towards the end of the transaction have to be acquired early on (in the growing phase). In a CAD environment, where transactions are of long duration, two-phase locking would considerably reduce concurrent access to data.

2. Locking protocols. Traditional databases are also quite inflexible in the data locking modes. Normally, they provide three types locks: read, write and exclusive. Locking is generally not communicative, i.e., the systems does not inform the user of the lock status of the object (e.g., whether it is free, or locked by someone, in which mode, etc.). Thus an user may wait indefinitely for a lock on an object without being informed in advance by the system of the lock status of the object. Also, an user holding a lock on object is not informed if anyone else is waiting for that object. Traditional databases do not support the notions of shared locks between collaborating users or allow two or more users to update the same object by simultaneously acquiring write locks on the object.

Two types of concurrency policies are generally adopted with two-phase locking:

(a) Conservative (or pessimistic) concurrency control: in this scheme, access conflicts are checked at actual access (or request) time. In the event of a conflict, the conflicting transaction stalls or is aborted.

(b) Optimistic concurrency control: in this scheme, conflicts are checked not at access time, but at transaction commit time. This actually allows more processes to operate concurrently than does the conservative policy; however, should there be a conflict there is the risk of losing a lot of work done in a transaction which may abort.

There is no notion of semantics-based or predicatewise (based on the satisfaction of a set of input and output conditions) concurrency control.

3. Version management. Version management of data is also not standard in most traditional database systems, and is not implemented for common business-type applications (for example, it does not make sense to have versions of someone's bank account). Most RDBMS supporting version management implement it only at the granularity of whole tables and not individual tuples. Thus versioning of big relations may prove to be expensive. The issue of
version configuration trees and dynamic binding of data to appropriate versions is largely unexplored.

4. **Nested transactions.** Conventional database systems do not support the concepts of nested transactions, where a transaction is hierarchically sub-divided into sub-transactions. There is also no support for grouped transactions, where multiple clients perform database operations and share data as a single transaction unit rather than across transaction boundaries.

5. **Communication.** Inter-client communication and update notification schemes are poorly implemented in most traditional database management systems. As a result, different users are not aware of each others' work, and are not notified when appropriate data are changed or updated. Locking schemes are also not communicative, as a result of which users are not aware of the lock status of data and may be in the dark indefinitely for lock requests to be granted.

6. **Conflict handling.** Conflict handling is generally poor in traditional transaction management frameworks. In the event of deadlocks or commit conflicts, generally, one or more transactions are arbitrarily aborted by the system without prior notification to the clients involved. The system does not take into consideration the semantics of the data or transactions or the urgency of certain transactions over others, etc. Since long CAD transactions may represent a great deal of work, an abort without notification may prove quite expensive.

Though the above techniques are adequate (and possibly even desirable) for financial applications, since CAD transactions are often of long duration, it is highly inappropriate for such work because it inhibits information sharing and result in reduced concurrency and intolerably long waits. Serializability is thus too limiting a criteria for such work.

Special considerations are therefore necessary for the design of a more flexible and efficient transaction management system which allows a group of designers to arrive at a complex design without being forced to wait over long periods, and enables collaboration among design groups.

### 4.4 A Model for a Collaborative CAD Environment

Automated design activities (CAD, CAE, CASE etc.) share a number of characteristics [15, 25]. The design effort is generally divided into several projects. Each project sub-division accesses certain pieces of information that are unique only to the sub-task, and certain other information that is shared with other sub-tasks. Thus certain design data is "global" (accessible to all) while other (more specialized data) is "local" (with various levels of access restrictions). In each project, designers further sub-divide the project into a number of sub-tasks, which may again be decomposed
into simpler units. Often this scheme represents a top-down hierarchical decomposition of a design activity.

As designers work on well-defined, fairly small sub-tasks, there is a far greater need for shared access to design data than amongst the projects at a higher level. Thus transactions need to cooperate and interact. Also, designers form groups to work collaboratively on a sub-task. This leads to the notion of cooperating grouped transactions. The functional subdivision of a design activity is shown in Figure 4-2.

For example, the design of a building, as illustrated in Chapter 1, can be discretized into a hierarchy of tasks as shown in Figure 4-3. The top level activities include: planning, estimation and cost analysis, architectural layout, structural design, electrification, sanitary engineering, fabrication, etc. Each of these activities may be decomposed into better-defined subtasks. For example, the structural design may consist of the design of superstructure and substructure, each one of which consists of the design of various components like roof, floors, footing, foundations, etc. Likewise, the architectural design consists of a skeletal layout of the building frame and the more detailed interior design of rooms, walls, hallways, etc. As illustrated by the example in Chapter 1, designers need to interact extensively at the lower level of sub-tasks. For example, the architect and structural engineers may work in a group for the design of the building skeleton. The structural and geotechnical engineers interact for the design of the building foundation. The structural engineer may simultaneously be a member of both design groups. Alternatively, if the design task is small, the architect, structural and geotechnical engineers may all work in a group for greater concurrency of design, as shown in Chapter 1.

The above functional design model can now be translated into a model for distributed CAD performed on a network of engineering workstations, as proposed in the DICE framework (Figure 1-6, Chapter 1). If we consider a piece of design activity as a high level CAD transaction which performs certain interactions with the database (or subdivisions thereof), then hierarchical discretization of a design task may be mapped into a hierarchy of discretized transactions. The design environment then consists of a number of high-level project transactions, each of which are are divided into nested sub-transactions, which in turn may be composed of several lower level transactions. The concept of nested transactions was introduced by Moss [29], and will be explained later. Several of these sub-transactions may be organized into cooperating groups. Each group may consist of several designers, each working on his/her personal workstation, and a designer may be involved in more than one group. This scenario is represented in Figure 4-4. Thus the CAD transaction framework is hierarchically organized in a tree with the leaf transactions representing design applications, and internal nodes representing groups or nodes of task sub-division.

The functional decomposition of the design transaction also results in a physical partitioning of the database into different local shared databases associated with each sub-transaction and design group. Each sub-transaction has its own data buffer pool derived from its parent transaction, which is inaccessible to other sub-transactions. Each transaction commits or posts its results to its parent at which point the data becomes visible to other sub-transactions derived from the same parent. These protocols for change propagation will be discussed shortly, and are introduced here for
Figure 4-2: Task discretization in a design project
Figure 4-3: Task discretization in design of building
Figure 4-4: Model of CAD transactions showing transaction nesting and grouping
presentation of our model. Each designer has a private database for doing work inaccessible to others (unless explicitly authorized to do so). Similarly, each group has its own local shared dataspace for active data sharing and design experimentation inside the group. Communicates with extraneous project databases takes place through the parent database. Thus, the whole database is discretized into a hierarchy of “local-global” databases (i.e. databases which are global to a set of groups, but local to the sub-task or sub-transaction). This scenario is illustrated in Figure 4-5. An example of database partitioning scheme for building design has been shown in Figure 1-7, Chapter 1.

To summarize our model for a collaborative CAD environment we may again stress the following issues:

- design transactions are sub-divided into smaller nested transactions representing specific design tasks;

- several transactions may form design groups for local data sharing for a specific area of design;

- for more active data sharing and interaction, for example during experimentation and negotiations, users participate in shared transactions;

- each design group maintains its own local shared database which is inaccessible to members outside the group; once changes inside this local database are finalized by mutual agreement, they are checked out to the parent database for sharing with other users outside the group. This ensures that only consistent designs that have been agreed upon by a group of designers (responsible for the work) are made visible to others. This is referred to as encapsulation of localized design effort;

- good communication and update notification and conflict handling facilities enable interaction between designers and keep them aware of each others’ work;

- version management of data keeps track of the evolution of design by preserving old data and keeping track of changes made to data. In addition it also enhances the concurrency of design by easing the bottlenecks caused by locking;

- the framework attempts to exploit design semantics wherever possible to allow for greater concurrency and flexibility by unconventional techniques, such as non-serializability.

The key features distinguishable are a nested framework of grouped transactions that encapsulate nonserializable data sharing and a provide for localized specification of application semantics to determine the validity of data. The issues enumerated above are discussed in detail in the following section.
Figure 4-5: Database partitioning for collaborative CAD
4.5 Transaction Management Functionalities for Collaborative CAD

In the previous section, we have discussed our model for a collaborative CAD environment. We now describe the various transaction management functionalities that are necessary for such an environment. The transaction management modules in an OODBMS have been enumerated in a previous section. In the following sections, we discuss each functionality in detail with emphasis on the enhancements that are necessary for supporting our model.

4.5.1 Interactive Transaction Framework

In our framework, CAD transactions are interactive (involve active human interaction). Since the nature of a particular transaction may depend on the results of another, they are mostly defined dynamically at run-time. Thus, the execution environment is interpretive. This allows designers to react to design changes by others interactively, to develop their applications incrementally and incorporate changes without application shutdown. This is very useful for dynamic schema evolution in object-oriented design [43], which allows incremental development and modification of class structures during design. For instance, the structural engineer may find it useful to modify a design methodology or add a new design class on the fly. In an interpretive environment s/he may do so without having to stop, recompile and relink the whole application.

All transactions are represented as persistent objects in the database and are tagged with an identifier, the id of the user who initiated it, and keep a record of the object buffer pool affected by the transaction.

4.5.2 Client, Data, and Transaction Record

In a collaborative environment it is necessary to maintain records of all the users using the system and their activities (or transactions). It is also necessary to maintain data records about the data in the database, e.g., the ownership, timestamp, and usage of different objects. In our framework, all users are represented as persistent objects which record the user’s id, objects accessed, and transactions initiated. Every object keeps a record of its ownership, timestamp of creation and last modification (with the id of the modifier), history of modifications, lock status, lock dictionaries, version set (version management will be discussed shortly) and a dependency list of all users who have accessed the object. This information is useful for keeping track of the design history and for inter-client communication. All transactions are also represented as objects and are tagged with an identifier, the id of the user who initiated it, the data (objects) accessed by the transaction and its final status. Keeping all these records incurs some performance overhead but it helps to keep track of design evolution and data usage by designers.

Some of the principal operators that are necessary for maintenance of the above
records are as follows (refer Appendix A for details of member functions of various transaction management classes):

- `register_user(user_id)`: this registers an user when s/he accesses the database;

- `register_object(user_id, time)`: record information such as creator, timestamp; create lock, update and history dictionaries, etc.;

- `add_dependent(user_id)`: adds the id of the user to the dependency list of an object;

- `delete_dependent(user_id)`: drops an user from the dependency list;

- `record_update(user_id, time, change_type)`: records an update on the object in the history of modifications;

- `register_transaction(user_id, time)`: records a transaction with timestamp and user who initiated it;

- `insert_trans_buffer_pool(object_id)`: insert object id in the transaction buffer pool;

- `delete_trans_buffer_pool(object_id)`: delete object id from the transaction buffer pool;

- `show_users()`: lists the users using the database at a given time;

- `show_dependents()`: lists the dependents of an object;

- `show_update_history()`: displays the update history of an object.

### 4.5.3 Lock Management

Traditionally, the intention of locking objects is to provide "clean" read or write access to data objects to a single user (or groups of users) and to control concurrency of parallel transactions. As has been explained earlier, two-phase locking ensures serializability and ensures correctness of data in the shared database irrespective of the nature of the transactions. However, this scheme also reduces concurrency and parallelism of design effort. Therefore, locking mechanisms must be flexible enough to allow multiple transactions to proceed concurrently without having to wait indefinitely for other transactions to complete. Object locking is intimately associated with communication modes which are discussed shortly. The other option for ensuring greater concurrency inspite of locks is that of version management, which will be discussed in a later section.
Types of locks

The following represents the minimal set of lock types and their functionalities (modeled after [18]):

1. **Non-restrictive read lock** (NR-R), which allows an user to read an object without prohibiting the access privileges of others. It is the least restrictive lock, and does not conflict with any other lock type. This provides a snapshot of the database at the time of request;

2. **Restrictive read lock** (R), which prevents other users from updating the object during the duration of the lock;

3. **Non-restrictive write lock** (NR-W), which prevents other users from acquiring a read or write lock on the object, but allows a NR-R lock;

4. **Restrictive write lock** (W) or **exclusive lock**, which prevents other users from accessing the object in any mode.

5. **Shared-write lock** (S-W), which applies only to transaction groups. It allows two or more clients to simultaneously hold write locks on the same object inside a transaction group, so that they may all update the object and be notified whenever that happens. This is useful for active data sharing and experimentation inside a grouped transaction.

Lock information management

The system maintains a record of the lock status on each object and the clients holding those locks. This information facilitates communication between clients in the event of a conflict.

Lock communication

All forms of locking are highly communicative in the framework, so that that users are fully aware of the lock status of any object. Inter-client communication is initiated for all activities, such as

- **lock requests** - notification is sent to all the users holding locks on the object regarding the nature of the lock requested and the id of the requester;

- **lock releases** - all users holding or waiting for locks on an object are notified of the lock release;

- **lock attainments** - all concerned users are notified of the type of lock granted to a new user;

- **lock waits** - all users whose requests have been queued are informed, along with information on who else is holding locks on the object;
• lock denial - in the event that a lock cannot be granted (e.g., deadlock situation),
  the requester is informed without aborting the request.

The different modes of lock communication are very important for enabling cooperative interaction between designers. For example, assume that the architect (A) wants to update the dimensions of a Beam object that the structural engineer (S) is in the process of designing, and therefore has a write lock on it. When A requests a write lock on Beam s/he is informed that S is probably updating it. Thus A does not have to wait indefinitely for the object, s/he can proceed with other work and will be automatically notified when Beam is released. Thus, A can then either delay the update, or contact S about the changes if they affect the design of S.

Lock request management

All lock requests are queued if they cannot be granted immediately, and the user is informed of the status of the request. If it conflicts with a lock granted to another user, the associated communication protocols would inform the locker about the request, and the requester about the conflict and the nature of the lock held by the other client. The user may then prefer to wait (and do something else), abort the request, send a message to the locker to release the object, or, if the object is versionable, derive a new version of the object from a desired version in the database. Lock requests are never aborted by the system without appropriate notification.

Locking granularity

Locks may be set at various levels of granularity, e.g.,

1. Object level, which is the most common and finest granularity of locking available;

2. Aggregate level, which includes locks on all members of a collection, e.g., a Set;

3. Classification level, which implies locking all the instances of a given class;

4. Version level, whereby a given or all versions of an object may be locked;

5. Composition level, which refers to the locking of component objects in a composite object.

OODBMS may also support locking at the level of

1. Class structures. In an environment where extensive dynamic schema evolution is possible, this implies locking of a class template itself, e.g., to prevent the class structure from being updated.

2. Schema or class hierarchies. This concerns locking part or all of the of the database schema or the hierarchy of classes.

Locking (and versioning) of classes and schema is an expensive process and is currently under investigation. A discussion of class lattice locking can be found in [17].
Lock tenure

The present framework supports a flexible protocol whereby locks may be requested and released anytime during a transaction. This enables a client to explicitly unlock an object in the middle of a transaction if it is requested by another client. Such a protocol violates serializability, but consistency of data may be ensured by other means, such as constraints, type semantics and nature of the transactions involved.

For example, assume that the architect is holding a read lock on a Floor object (for the purpose of doing interior layout let's say), and the structural engineer (S) wants it for design (or update) (see Figure 4-6). Since S will update attributes of Floor (such as slab reinforcement) that will not affect A's work, A can release the lock on Floor upon request from S, and later read the object back from the database after S is done. In A's perspective the object has undergone no change, despite the fact that S updated it. Thus, in this case the non-serializable schedule does not alter the "correctness" of data seen by A or S. Non-serialized "correct" schedules such as these are common in engineering design (as contrasted to the domain of financial applications), and therefore need to be exploited for greater concurrency. These typically occur in cases where two or more different users are modifying different perspectives or facets of the same object that have no effect on each others' work. (An alternative scheme for solving this problem is by versioning, which will be discussed shortly).

Lock management operators

The principal operators that are necessary for various lock management functionalities are enumerated below:

- `lock_entity(entity_id, locktype, time)`: places a lock on an entity, where entity could be either an individual object, a collection, a version set, composite object, or class; also record timestamp; also places a log in the lock dictionary of the entity, and sends notifications to all current lockers;

- `unlock_entity(entity_id, time)`: releases the lock on an entity; updates the lock dictionary appropriately, and informs all current lockers;

- `modify_lock(entity_id, new_locktype, time)`: upgrades the locktype on a locked entity, and notifies appropriate users;

- `add_locker(user_id, locktype, time)`: add a new locker to the lock dictionary of an object; implicitly called when lock is granted;

- `remove_locker(user_id, time)`: remove an user from the lock dictionary of an object; implicitly called when lock is released;

- `is_locked(entity_id, locktype)`: reports whether an entity is locked or not in a given mode;

- `show_lock_status(entity_id)`: displays the lock status of an entity;
Figure 4-6: An example of non-serialized "correct" schedule in building design
• `show_lockers(entity_id)`: displays all lockers of the entity;

• `notify_locker(entity_id, other_locker, message_string)`: sends a message to a specific locker;

• `notify_all_lockers(entity_id, message_string)`: sends a message to all current lockers;

• `queue_request(entity_id, user_id, locktype, time)`: queues a lock request on an entity;

• `detect_conflict(entity_id, time)`: detect a conflict in concurrent lock requests, and identify the conflict set of users and locks;

• `notify_lock_conflict(entity_id, conflict_set, time)`: inform all users in the conflict set that their lock requests are in conflict;

• `deny_request(entity_id, user_id, locktype, time)`: deny a lock request due to conflict or deadlock, with appropriate notification;

• `dequeue_request(entity_id, user_id, locktype, time)`: allows a lock request to be dequeued;

• `grant_request(entity_id, user_id, locktype, time)`: grant lock to request on queue; this is accompanied by notification to appropriate clients.

### 4.5.4 Deadlock Management

Normally, the system will not grant locks resulting in a deadlock situation because such lock requests will be queued due to access conflicts. A waits-for graph is used to determine cycles in lock queues. When a cycle is detected, the newer conflicting lock requests are dequeued and the user(s) informed. The system will attempt to avoid a transaction abort as far as possible.

The additional operators that are necessary for deadlock management are:

• `build_waits_for_graph()`: constructs the waits-for graph for every lock request;

• `detect_cycle()`: detects a cycle in lock request queues, and identify the conflict set of users and entities;

• `notify_deadlock_conflict()`: notifies the appropriate users regarding deadlock;

• `abort_request()`: abort one or more lock request, after proper notification.
4.5.5 Communication

One of the drawbacks in many of the current design database environments today is the absolute lack of communication facilities between users. Clients are often not aware of each others' developments, and this often leads to a lack of coordination, reduced parallelism, a considerable waste of time and resources, and sometimes even faulty designs due to misinterpretations of data. Effective inter-client communication is of paramount importance in cooperative CAD work since it facilitates active exchange of information, better synchronization of work across design interfaces and greater concurrency, all of which are essential for collaborative development. The following types of communication facilities are available in this framework:

1. **Lock communication modes.** When a client requests a lock on an object that is already locked in some form by another user, the locker is informed of the lock request, and the requester is notified of the lock already in place. When an user acquires or releases a lock on an object, other lockers are informed of the same.

2. **Update communication modes.** For every object, the system maintains a dependency list of all users that have accessed the object for reading or writing. When the object is subsequently updated or changed, all affected users are instantly notified of the nature of the change and the id of the client who made the change. This allows designers to react appropriately to the changes. For example, a designer who obtained a snapshot view of an object from the database while it was being updated by someone else, may now request the latest copy from the server and check to see if the results of prior work should be changed.

Update communication modes have been illustrated in the example in Chapter 1. When the architect (A) changes the specifications of certain building components (that the structural engineer, S, has read earlier), S is immediately notified of the changes and the nature of the change. Thus S may look into the changed components and verify their structural integrity, or redesign them. This timely notification is useful in several ways:

- it appraises designers of changes in design they may have been affected by;
- early notification of changes may help to reduce the amount work that has to be redone. For example, if S had completed only about 25% of his/her design by the time A specified changes, then S has to redo only a quarter of the work, and complete the rest with the new set of specifications. If changes are not notified promptly (as is the case in actual design practices today), then rework may be rampant;
- notification of changes also help to detect inconsistencies in design, particularly across design groups. Often designers are not aware of how designs are changed or misinterpreted by designers in other groups. A tragic implication of this lack of awareness may be seen in the Hyatt Regency disaster [28]. A detailed analysis of this problem can be found in [44, 43].

74
3. Conflict communication modes. In the event of a conflict between two or more transactions (for example, a deadlock or commit failure), the system does not abort the transactions, rather it raises exceptions notifying the conflicting clients of the nature of the conflict. It is assumed that the clients will interact among themselves to resolve the conflict appropriately. The system-controlled default is to abort the newer conflicting transactions and to send notifications to the relevant clients.

As an example, consider the case where two structural engineers are concurrently designing two different beams which share a common connection. Although design of the beams may be performed independently of each other, each engineer, in the process of having write-locked the beam objects will have implicitly locked the corresponding connection object as well. Assuming optimistic concurrency control, a simultaneous attempt to commit these beam objects will therefore result in a conflict. However, if both users are notified of the conflict, all the work may be saved by agreeing to commit one after the other (which may incur only a small delay that may be worthwhile) rather than together. A traditional system would have aborted one of the transactions arbitrarily, thus resulting in loss of work (assuming that the work had not been stored somewhere).

4. Negotiation modes. Often designers are faced with inconsistent design solutions across design interfaces and need to negotiate for a mutually acceptable solution (for example, trying to merge two different design versions of an object). Currently, clients may send messages to each other in the form of X-Windowsgrams; the present framework will eventually support a multimedia negotiation platform where video, images and hypermedia text help to expand the bandwidth of communication between clients.

Communication management operators

The principal operators required for communication management are as follows (some of these have already been listed in previous sections on client, data, and transaction record, lock management and deadlock management):

- `send_message(user_id, message, time)`: sends a message to a particular user;
- `send_message(user_id_list, message, time)`: sends a message to a list of users;
- `show_lock_status(entity_id)`: displays the lock status of an entity;
- `show_lockers(entity_id)`: displays all lockers of the entity;
- `notify_locker(entity_id, other_locker, message_string)`: sends a message to a specific locker;
- `notify_all_lockers(entity_id, message_string)`: sends a message to all current lockers;
- `notify_lock_conflict(entity_id, conflict_set, time)`: inform all users in the conflict_set that their lock requests are in conflict;

- `notify_deadlock_conflict()`: notifies the appropriate users regarding deadlock;

- `notify_commit_conflict()`: notifies affected users of the nature of the conflict;

- `update_notify(entity_id, update, time)`: notifies all users in the entity dependency list of an update, including versioning changes

- `inform_dependents(entity_id, message, time)`: send a message to all dependents of an entity;

- `arrange_negotiation(user_id_list, time)`: arrange for conferencing or negotiation between users, for example a zephyr instance.

### 4.5.6 Version Management

Version management allows data to be stored as a series of evolutionary changes to objects instead of overwriting the old copy when data is updated. Thus, representations of the same object may exist in the database as different versions of the object, each with a different content. The set of versions of a versionable entity is called its version set.

#### Versions and concurrency

Version management is essentially a data management issue. It will be discussed briefly in view of the role it plays in enabling greater concurrency. Versioning of objects significantly enhances parallelism of design effort. If an object is locked by one client, other clients may derive a new version of the object and continue working on it in their private dataspaces. This alleviates the problem of having to wait (often over long durations) for one client to release an object before others can use it. This is particularly useful in situations where different users work with different perspectives of some large multifaceted object. Instead of accessing the object in a serial manner (one user after another), different designers may concurrently work on their own versions and finally merge their designs together (merging of versions involves a negotiation framework we have discussed earlier). Versioning also allows multiple users to concurrently write or update the same object without violating database consistency, since all versions are stored in the database and no existing objects are overwritten.

As an example of version-aided concurrency, consider the problem we considered in a previous section, where the architect (A) held a read lock on the Floor object that the structural engineer (S) wanted to design. If the object is versionable, then S can check out a copy of the object to his/her private dataspace, and complete its design. When the completed design is checked back into the shared dataspace, a new version of the Floor object is created (see Figure 4-7). The two versions differ in the completeness of the design of the slabs. Since the new version has all the information
of the first one together with the design, the older version may be rejected after A is done. Thus versioning enhanced concurrency of design without compromising data consistency.

Versions of objects also help to keep track of the evolution of design since objects store their version history. Transaction histories maintain records of specific versions of objects used in design. Merging of different versions represent negotiation and approvals of a common design at different stages. Maintenance of design history is very important for collaborative work where several designers update the database frequently. In the event that a design appears to be faulty at any stage, it may be possible to rollback the design to some valid data state by retracting to a previous version.

Versions and locking

It would appear [40] to some that versionable objects obviate the necessity for locking, since a new version of any locked object can be created and checked back into the database. There are various reasons, however, to believe that versioning complements locking as a concurrency control mechanism rather than precluding it.

The above versioning scheme allows unchecked access and update privileges to all versionable objects. Unrestricted privileges may, however, be undesirable in certain situations, just as locking may be necessary in others. Versioning of objects is expensive in terms of storage, record-keeping and processing overheads, and versions should not be created unless deemed necessary. A lock on an object denotes the data is being used by someone for some useful purpose or is otherwise not intended to be accessed at will. In such cases, another designer also needing the object for update may create a new version. If however, the object is not locked by anyone, versioning may not be necessary at that point. If there were no locking, there would be no way to tell whether an object is being used by someone and in what mode (read, write, etc.). In the absence of locking, we would either end up in creating a proliferation of versions (each designer would create a new version every time s/he needs an object) or a plethora of commit conflicts (several users would try to update the same object without knowing in any way that others want to update it too!). Further, in our framework, as in others, user data access records and client-data dependency records are created during the process of locking. Thus locking provides a mechanism for recording data usage by multiple users, and also provides a criteria by which new versions of objects may be created.

Further, there may be situations during which one would not want certain data to be read or any new versions created from them for various reasons. A good example may be taken from the domain of CASE, where the source code is periodically "frozen" from further development. In these situations, specific objects may be locked in the exclusive-lock mode to prevent all forms of access to these objects only rather than closing off the whole database. Thus locking provides a mechanism for complete isolation of certain pieces of data, and for control of version proliferation of stable objects.

Objects may also be locked for monitoring purposes. A non-restrictive read lock


**Architect**

- Read_lock (Floor)
- Read (Floor)
- Interior layout, etc.

**Shared Database State**

```
F0
```

**Structural engineer**

- Request write_lock on Floor
- Check out as a version of Floor
- Design Slab (Floor dimensions unaffected)
- Update Floor (post)

```
F0 -> F1
```

- Read_lock(F1)
- Check to see if work affected

```
communication
```

- Agreement on design
- Release version F1
- Delete F0 (redundant)

---

**Figure 4-7**: Example of enhanced concurrency due to versioning of objects
effectively places a monitor on the object (without restricting access privileges of others), so that the locker is informed every time the object is accessed or used by others. For example, database administrators may like to monitor the use of system objects by locking them in a communicative mode.

The utility of data versioning extends beyond the necessities of concurrency control. Versions are created every time data is updated, irrespective of whether the object is locked or not. Versions are most useful for recording incremental changes to data and for allowing mutable versions to coexist in the database. Version management is also an interesting data modeling issue (independent of locking and transactions) and should be viewed from these perspectives rather than strictly from the point of view of concurrency management.

Version trees and hierarchies

A version set of an object is a tree structure with each node corresponding to a version and linked to other nodes by predecessor or successor links. Version trees may either be linear or branching as shown in Figure 4-8. Linear versions are created by successive derivations of a new version from the newest version. For example, a linear version of the House object is created in our example when the architect specifies a new set of specifications. Branching versions are created by creating two or more different versions from the same base version (for example, simultaneously by different users).

Check-out and check-in

Check-out refers to creating a copy of an entity in the global or shared dataspace in an user's private dataspace. When the user updates the object and posts it back to the global database, it is referred to as check-in. Check-in and check-out are closely related to the versioning mechanism. Users may check out specific versions of an object into their private dataspaces. When the objects are checked back in, a new version of the object is automatically derived from the base version and added to the object's version set. Since the checked-in version of the object is only a copy of the object in the global database, the designer may work on it (in the private database) as long as it is needed without restricting the access privileges of other users to the object in the global database. This enhances the concurrency of effort between multiple designers working on the same object.

Figure 4-9 shows how a linear version of an object is created in the database due to a single user check-out. Figure 4-10 shows the creation of branching versions due to multiple users checking out the same version of an object separately and later checking them back in. As with all update procedures, check-ins also generate update notification messages to dependents.

Examples of versioning have been presented in the example in Chapter 1. When the architect (A) checks in the House object with new specifications, a new version H1 is derived from the original object (H0). A simultaneous check-out of House by two different structural engineers for example (for designing different parts of the
A linear version set

Parallel and branched versions

Figure 4-8: Linear and parallel versions
Figure 4-9: Check-out and check-in (single user)
Global database

State 1

Check-out

Modify

Private dataspaces

Check-in

Modify

User 2

(branched versions)

State 2

Check-out

Check-in

User 1

Figure 4-10: Check-out and check-in (multiple users)
house) would have resulted in parallel versions of the object.

Version granularity

Object-oriented databases may support three types of versioning capabilities:

1. Versions of objects (instances). Instances of versionable classes (or subclasses thereof) may have versions. Versions of such objects are created whenever they are changed or updated; instead of overwriting the existing object, a new object (with the changed attributes) is derived from it and added to the version set. A versionable object maintains detailed record of its history, including the version number, parent and derived versions, a timestamp, and the id of the client (or transaction) that created the version. It may even be possible for creators of versions to record the design rationale involved in the object updates. Users may also explicitly create or delete a version of an object (provided they have appropriate privileges).

2. Versions of classes. This refers to versioning of classes when their definition or structure is changed. When the structure or composition of a class is updated at runtime, there occurs a problem of migrating these changes dynamically to existing instances of the class (and those of all subclasses of the class). This may be a very expensive and unpredictable operation, and most OODBMS either disallow dynamic class modification, or let the existing instances retain their original structure and behavior. Versioning of classes is a viable solution to this problem. When a class is redefined, a new version of the class is created, while the old definition co-exists in the database with its set of instances (called the extension of the class), Figure 4-11. This may be quite useful in situations where design classes are changed frequently for the purpose of experimentation or testing. It allows different users to work with their own version of the class definition and the associated set of objects.

Versioning shown in our example all concern versions of instances or objects. As an example of versions of a class consider the following. In our schema for a building (see Figure 1-5), the Beam is a part of Room. Alternatively, Beam could be made a part of Slab for the purpose of design. For this purpose, we need to add attributes to the class Slab which record the number of beams supporting the slab and their identifiers. Similarly, a new attribute needs to be added to class Beam to indicate the id of the slab it is a part of. Appropriate new methods will also have to be added to the two classes to take care of these changes. If these changes are made at run-time, then existing instances of Slab and Beam will have a data structure and behavior different from those of the new one. Preserving the old definitions of these classes in terms of versions (rather than overwriting them) allows these old instances to coexist with the new instances and be used in the proper context.

3. Versions of schema. An even more complicated problem is that of changes in the structure of the class hierarchy or the database schema at runtime, for
Figure 4-11: Version management of classes
example, addition or deletion of classes, changes in the inheritance specifications, etc. Though this is infrequent in a well planned design process, it may happen when the nature or scope of the design undergoes a major change. In such an event, a new version of the whole schema may be created with its own set of instances. It is possible that quite a significant number of objects may be unaffected by schema versioning, and hence may be shared between different schema versions. Elaborate schema versioning schemes and algorithms for mapping an object to the appropriate schema to which it belongs can be found in [21]. Schema versioning has not been implemented in our framework.

4. Versions of composite objects and aggregates. Use of composite objects [20] are commonplace in engineering CAD. Composite objects and aggregates (e.g., collections) as the unit of version control is currently investigation. In composite objects, broadly speaking, there may be two types of changes:

(a) changes in the containing object, which percolate down to the component objects, and is called change percolation; and

(b) changes in the component objects which propagate to other component objects and to the parent object, and is called change propagation.

Both types of changes lead to problems of version configurations of the composite objects. For instance, consider a change in component C1 of the composite object C0 shown in Figure 4-12. Two (of several) possible version configurations of C0 are shown in the Figure. In one, two versions of the whole composite object is created, each containing a different version of C1. In the other, a branching version tree is created at the component level. There may be several other configuration possibilities depending on the extent of data sharing between the parents and components, the depth of the component hierarchy at which change occurs, etc., and no one strategy is likely to be optimal for all composite objects for all possible types of changes. The choice of a particular configuration may be dictated by performance or storage overheads. These issues are currently under investigation [49].

Consider for example, the example of the Car composite object introduced in Chapter 3 (Figure 3-2). A change in the attributes of an important high level component such as the Drivetrain is likely to affect the designs of all other major components, such as Body and Utilities, and as in real life, and we would essentially have a new model of the car. Thus, for such a change, creating a new version of the whole composite object is justified (Figure 4-13). A change in a lower level and relatively isolated component such as the size of Knob is unlikely to affect other components, hence a local version of the object may be created at the component level (Figure 4-14). The versioning scheme must therefore take into account the semantics of the changes, and the extent of their effects on the composite object as a whole.
Figure 4-12: Composite object configurations
Figure 4-13: Example of composite object versioning showing change in significant component
Figure 4.14: Example of composite object versioning showing change in isolated component
Version management operators

The main operators that are necessary for version management are as follows (for details of member functions of version management related classes, refer Appendix A):

- `create_new_version(entity_id, time)`: this implicitly creates a new version when a versionable object is updated; this may be type specific for composite objects, collections, etc.;

- `derive_new_version(parent_version, time)`: explicitly derives a new version from a given version; takes into account timestamp, creator, design rationale, preceding version links etc.;

- `display_version(version_id)`: displays a specific version;

- `display_all_versions(entity_id)`: displays all versions of an entity;

- `display_recent_version(entity_id, time)`: displays the most recent version in the version set

- `set_default_version(version_id, time)`: sets a given version as a default version accessed by a user;

- `get_default_version(version_id, locktype, time)`: returns the default version of an entity, and locks it in a given mode;

- `get_specific_version(version_id, locktype, time)`: returns a specific version from the version set;

- `get_all_versions(entity_id, locktype, time)`: retrieves the entire version set;

- `get_predecessor_version(version_id, locktype, time)`: gets the parent version of a specific version;

- `get_succeeding_versions(version_id, locktype, time)`: gets all the succeeding versions of a specific version;

- `display_specific_version(version_id)`: displays a specific version;

- `display_all_versions(entity_id)`: displays all versions of an entity;

- `display_number_of_versions(entity_id)`: displays the number of versions in the version set.

- `display_recent_version(entity_id)`: displays the most recent version in the version set

- `delete_specific_version(version_id, time)`: deletes a specific version, reconfigures the version tree;
- **delete_all_versions**(entity_id, time): deletes the version set of an entity;

- **delete_succeeding_versions**(version_id, time): deletes all the succeeding versions of a given version;

- **delete_predecessor_version**(version_id, time): deletes the parent version of a given version;

- **show_difference**(version_id1, version_id2): displays the differences between two versions of an entity, attribute by attribute;

- **check_out**(version_id, database, locktype, time): checks out a copy of a given version into the named database;

- **check_in**(versionable_object, database, lockflag): checks back a given object into the named database, with the lockflag indicating whether the lock placed at checkout should be released or not;

- **merge_versions**(version_id1, version_id2): an application or type specific function that merges two versions into one version; may involve more than one user in an interactive or negotiative session, or may be automated by code;

- **display_version_tree**(entity_id): graphically displays the version tree.

### 4.5.7 Conflict Handling

The current framework supports robust schemes to handle conflicts between clients and their database operations. In traditional databases, the system aborts (often arbitrarily) one or more conflicting transactions in order to resolve the conflict, often without informing the affected clients. This may result in crashing of the application or otherwise loss of significant amounts of work. The philosophy of conflict resolution in the present framework is to avoid system-dictated aborts at all costs, and effective communication is the key factor in implementing this scheme. In each case, the clients concerned are first notified of the nature of the conflict and all the clients involved in the conflict, so that each user is fully aware of the situation. The clients communicate with each other to resolve the conflict in an agreeable, semantically reasonable manner. For example, in a CAD environment, often certain tasks are more important or urgent than others, or it makes sense to have certain transactions precede others. The present scheme allows such jobs to proceed unimpeded. In addition, it may be possible for the users to avoid aborting their transactions and simply attempt to commit later.

There may be several types of conflicts in a CAD environment, such as:

1. **Lock conflicts.** These occur when two or more clients request incompatible locks on the same object at the same time. In the event of such a conflict, lock requests are queued and all clients involved are notified of the status of the lock on the object(s). The clients may communicate to withdraw their requests in favor of others, or request the lock holders to relinquish their locks, or wait till
locks become available, or derive a new version of the object. For example, two structural engineers in the process of design may happen to request write locks on the same Column object for design. The situation is simply resolved by one designer withdrawing his lock request, so that the other can go ahead with the design.

2. Commit conflicts. These occur when two concurrent transactions are in conflict according to any specified criteria. The system informs the clients involved who may then confer with each other to resolve the problem. For example, they may abort their transaction(s), delay commitment, or do something else to handle the situation in a semantically appropriate manner. This has been illustrated earlier in the context of our example, where two structural engineers want to commit the designs of two different Beam objects sharing a common connection. The engineers are notified of the conflict, whereby they communicate to find out the reason for the conflict (i.e. common connection). Once that is known, it becomes clear that the commits are actually compatible and the order is immaterial; hence the transactions may be committed in sequence.

As another example of semantics based conflict resolution, consider the case where the architect (A) is trying to post a set of dimensional constraints on various House components when the structural engineer (S) is trying to commit his design on some of these components. A conflict occurs, and A and S are notified whereby they inform each other about their commit objectives. In this case, clearly, the architect's transaction should precede S's, because the designs have to be validated against the set of constraints set by A. Hence, the constraints need to be committed to the database first before the design can be posted.

3. Design conflicts. These are application-specific conflicts that arise due to violation of design constraints or disagreement between designers. Since in object-oriented methodology, data updates take place through methods or functions of objects, violation of constraints may be easily detected. In such cases, communication handler objects may be instantiated from the application to send notifications of such violations to appropriate users. In the event of disagreement over design specifics, a constraint based negotiation framework will have to be used for resolving conflicts.

Conflict handling operators

The operators for handling lock conflicts have already been enumerated elsewhere, and will not be discussed again. Operators that are required for handling commit conflicts are as follows:

- `detect_commit_conflict(trans_buffer_pools)`: detects conflicts in concurrent object updates by comparing the transaction buffer pools of different transactions involved, and obtain the ids of the users concerned;

- `notify_commit_conflict()`: notifies affected users of the nature of the conflict;
- `queue_trans_commit(trans_id)`: queues the commit of a conflicting transaction upon specification from the initiator or other authorized user;

- `abort_trans(trans_id)`: aborts a given transaction from the conflict set after notification to appropriate client, or upon specification from the user;

- `trans_commit(trans_id)`: commits a specific transaction from the conflict set;

- `commit_queued_trans(trans_id)`: allows an user to attempt to commit a transaction that has been queued due to conflict;

*Design conflicts* are detected by application specific code, such as data access or update member functions of a class, which could take into account the validity, integrity, referential and other constraints set on the design data. Once a conflict is detected, appropriate notifications are sent to the clients concerned using the conflict detection operators discussed earlier. Also, constraint violations may be recorded whenever a designer chooses to violate a constraint, and the designer setting the constrained is notified. Design conflict handlers may also need to initiate a negotiation mode for resolving conflicts between multiple users. The application independent operators may be enumerated as follows:

- `notify_design_conflict(user_id_list)`: notifies all the users involved in the conflict;

- `record_constraint_violation(user_id, entity_id)`: records a violation of a predefined constraint on an entity by a designer;

- `notify_constraint_violator(user_id, entity_id)`: notifies the designer who violates a constraint;

- `notify_constraint_setter(user_id, entity_id)`: notifies the designer who set the constraint;

- `arrange_negotiation(user_id_list, time)`: arrange for conferencing or negotiation between conflicting users, for example a zephyr instance.

### 4.5.8 Transaction Nesting

The notion of nested transactions was proposed by Moss [29]. As discussed earlier, this allows a transaction to be sub-divided into smaller sub-transactions, which may be further sub-divided into nested transactions. In an engineering CAD environment, transaction discretization and nesting often reflects the hierarchy of design task division.

Transaction nesting allows a big transaction to be broken down into smaller, more controlled units, each specializing in a specific sub-task. We have already cited examples where the top level transaction for building design may be sub-divided into transactions concerning architectural layout, superstructure and substructure design, etc., which may then be further decomposed into more specific tasks. The number of objects that are affected by each transaction as a result of discretization of a big
transaction into smaller units is also small, implications of changes to those objects are localized, and it is easier to undo undesirable changes. Thus nested transactions provide smaller units for transaction commits, recovery, and rollback.

Nested transactions abide by several protocols. Locks are acquired through the parent transaction, and a nested transaction may hold locks of strength equal to or less that those acquired by its parent. Each nested transaction has its own private data pool, which is often a private partition of the database. Thus functional sub-division of tasks also lead to structural partitions of the database. Nested transactions upon commit post their data pool to the data pool of the parent transaction (which may be a database area shared by a few users) and not directly to the server.

The overall hierarchy of functional decomposition (along with groups, discussed next) and nesting provide a mechanism to achieve greater degree of concurrency and coordination in complex engineering design.

The operators required for transaction nesting include those required for initiating a transaction inside the scope of another transaction, creating a local dataspace for the transaction buffer pool, committing, aborting or checkpointing transactions, etc. These are enumerated below.

- \texttt{start\_transaction}(trans\_id, parent\_trans\_id, user\_id, time, database\_name): initiates a nested transaction within the scope of a parent transaction and creates a local dataspace with a given name; this also registers the transaction, with timestamp and ownership information;

- \texttt{create\_nested\_database}(list\_of\_user\_ids, database\_name): creates a local shared database; implicitly called when a nested transaction is initiated;

- \texttt{commit\_transaction}(trans\_id, time): commits a transaction;

- \texttt{abort\_transaction}(trans\_id, time): aborts a given transaction;

- \texttt{show\_parent}(trans\_id): displays the id of the parent transaction;

- \texttt{show\_children}(trans\_id): displays the ids of the sub-transactions;

- \texttt{show\_siblings}(trans\_id): displays the ids of all siblings of a given transaction;

- \texttt{display\_trans\_hierarchy}(trans\_id): graphically displays the nested transaction hierarchy;

- \texttt{display\_data\_pool}(trans\_id): displays the data pool of the transaction.

4.5.9 Transaction Groups

A transaction group [15] is a process that is the locus of control for a set of cooperating transactions. The motivation for grouped transactions may be stated as follows. Often, several designers would like to collaborate on a specific piece of design, and be able to interact extensively. Design involves iteration and experimentation, and thus the design objects go through a several transient phases before a stable state
is reached. If these transient states are committed to the global database and made visible to others, a chaos may soon result. It would therefore be useful to restrict the visibility of these changes only to the design group concerned. This leads to the formation of a separate design group with a local shared database.

This is exemplified in Figure 4-15. Assume users U1, U2, and U3 are using the database; U2 and U3 are designers collaborating on the design of an object, O, while U1 is a documentation monitor, which updates the documentation on object O every time it is updated. When U3 does a transient update of the object to notify U2, U3 also gets notified and records the update. Thus when U2 and U3 are making design changes between themselves, U1 records several inconsistent or incomplete versions of O. U2 and U3 therefore need to form a design group whose behavior should be isolated from other non-cooperating users as shown in Figure 4-16.

Like nested transactions, transaction groups may be nested to any arbitrary depth to compose a transaction group tree (Figure 4-17). The transactions in a group interact with each other and with its parent, which in turn interacts with other transactions at its level. A transaction group defines a protocol for its children, an input protocol and participates in a protocol with its parent, an output protocol. At each level, the transaction groups must follow the input and output protocols set for the group.

There are several input and output protocols that groups may abide by, and some of these may be defined by the users. These include locking protocols, change visibility protocols, etc.

Each group follows a lock filter protocol, whereby a group requests locks on database objects on behalf of its members. The members may only acquire a lock of equal or weaker strength than that acquired by the group. For example, a member transaction will not be granted a write lock on an object for which the parent group has acquired only a read lock. However, the parent group may communicate this requirement of its member to the server (or to its parent) and attempt to acquire a write lock for the group.

Transaction groups may also have data visibility protocols (also called object propagation rule). As mentioned before, the parent group checks out objects into its private group dataspace for its member transactions for active dynamic sharing and experimentation. None of these changes are visible outside the group until the member transactions jointly commit the objects and release it to the parent. This helps to shield one group of transactions from the undesirable effects of other unrelated ones. The parent transactions then share these objects with other siblings and in turn release a stable version to their parent. The communication protocols discussed earlier play an important role in all forms of interaction.

Since operations inside a transaction group are not visible outside the group and do not affect the extraneous database, consistency conditions on data in the group buffer pool may be significantly relaxed. This allows for interactive experimentation and data sharing between multiple users. The transaction schedules do not have to be serialized since the transient inconsistent states are not visible outside the group, provided the designers are aware of the implications of each others' work. When the designers agree on a common design, a stable version of the object is checked back
Figure 4-15: Motivating example for transaction grouping
Figure 4-16: Solution by transaction grouping
Figure 4-17: Transaction group tree
into the datapool of the parent transaction for sharing with other users.

Figure 4-18 shows an example of grouped transaction processing in building design. Let us consider the interaction involved in design of the building superstructure and substructure. The architect (A) and structural engineer (S) collaborate on the design of the superstructure, while the geotechnical engineer (G) performs the substructure design. The design link between the two groups is the load borne by the the columns and their configuration on plan. A and S form a design group (A-S) which is nested inside the group (A-S-G) for co-cooperatively designing superstructure elements like beams, floors, columns. The A-S-G group holds a non-restrictive write lock on Column objects (to allow update) while G holds non-restrictive read lock to allow reading and update notification. G does not need to know the intermediate steps during the iterative design, but is primarily interested in the final column design. When the column design is complete in the A-S group, it is committed to the data pool of the A-S-G group so that G is notified of the update, and can access the required data from there.

One of the marked advantages of this approach is that object management becomes a hierarchical process, since low level transactions acquire their locks through their higher level parents and not directly from the server. As one goes up the CAD transaction hierarchy, the number of transaction units interacting with the server decreases and thus enables transaction management at a higher and simpler level of abstraction. At each level, the transaction scheduler has to deal with a limited set of transactions, which is unlike traditional models where all transactions constitute independent units and compete with each other at the server for database access.

Operators for grouped transactions

The following operators are required for implementing transaction grouping:

- `start_group_transaction(trans_id, list_of_user_ids, parent_group, database_name)`: this registers a grouped transaction for the users concerned, and creates a local shared database accessible to these users;

- `create_group_database(list_of_user_ids, database_name)`: creates a local group database; implicitly called when a grouped transaction is initiated;

- `commit_group_transaction()`: commits the group transaction, and deletes the private dataspace; a commit takes place after design agreement between the users in the group;

- `abort_group_transaction()`: abort the transaction, thereby rejecting all changes;

- `show_group_parents(trans_id)`: displays the ids of the parent transactions;

- `show_group_children(trans_id)`: displays the ids of the children transactions;

- `show_group_siblings(trans_id)`: displays the ids of all siblings of a given transaction;
Figure 4-18: Example of grouped transaction in building design
• `display_group_trans_hierarchy(trans_id)`: graphically displays the grouped transaction hierarchy;

• `display_group_data_pool(trans_id)`: displays the data pool of the transaction;

• `display_group_members(trans_id)`: displays the ids of the users in the group;

• `notify_group_members(message)`: sends message to all the group members;

• `set_group_protocols()`: this is an user defined application specific code for setting the lock, data visibility protocols, etc.;

• `join_group_transaction()`: allows an authorized user to join a transaction group;

• `quit_group_transaction()`: allows an user to quit a transaction group.

Operators for check-in and check-out, discussed earlier, are used for moving objects between parent and child databases.

### 4.5.10 Shared Transactions

Shared transactions represent a mechanism for allowing the greatest flexibility and freedom of interaction between designers. In this case, two or more users participate in a single, atomic, multiuser transaction [35]. This allows for unrestricted data sharing between multiple users without having to cross transaction boundaries, which may be useful for negotiation processes. For example, users participating in a shared transaction for a specific design may all acquire shared write locks on the design object concerned. Each user may update the object locally, and each update sends notifications to other users who respond to them appropriately. The transaction schedules do not have to be serialized, since the transient inconsistent states are not visible outside the group. When the designers agree on a common design, a stable version of the object is checked back into the parent database for sharing with other users. This promotes a greater freedom of interaction between multiple users than having to go through individual locking, transaction initiation and commit cycles. This is illustrated with an example in the following section.

As an example of a non-serialized schedule and inter-client communication in a shared transaction, consider a simple beam design example shown in Figure 4-19.

The clients involved in this activity are an architect (A) and a structural engineer (SE). They participate in a common transactions and share the Beam object (to be designed) in a local dataspace. The interactive design proceeds as follows:

• A specifies the dimensions (depth and width) of the beam and writes the object to the transaction buffer pool. Thereafter, A proceeds with other architectural work.

• SE is instantly notified; he reads the object from the database and performs a preliminary design. The dimensions are found unsuitable, hence SE changes the beam dimensions and writes the object, sending a message to A that Beam has been updated.
Figure 4-19: Cooperative interaction in a shared transaction
• A is notified of the change in Beam, and reads it from the database. If the proposed dimensions are agreeable, SE is notified and design is completed. Otherwise, A suggests new beam dimensions along with a set of constraints on the values and writes the object back to the database.

• SE is notified of the posting, reads in Beam, and redesigns it. If the constraints are satisfied, then the design is complete. If not, SE and A may have to negotiate for a mutually agreeable solution.

• When the design is complete, the object Beam is posted to the global database to be shared with other users outside the group.

The following points of interest may be noted in the example described above:

1. There is only one transaction in the above process in which both users participate, and all interactions occur inside the scope of this transaction.

2. The transaction schedule is non-serialized, since both A and SE read the object after they have written it.

3. Participants A and SE actively communicate with each other regarding the status of the design and are aware of each others’ developments.

4. All intermediate changes to object Beam are made inside the scope of the shared transaction data pool (or dataspaces) and are invisible outside. When the design is finalized, the object is posted to the global database, and all other users are then notified of the changes. Thus, external users of Beam are unaffected by these transient changes, and data is never in an inconsistent state in the global database.

5. It is possible for the participants to do other work (as part of other groups) during this design process, since each is notified whenever there is some change in the objects concerned and need attention. This promotes parallelism of design effort and increased productivity.

Operators for shared transactions

The set of operators that are required for implementing shared transactions are similar to those for nested transactions, and are enumerated below:

• start_shared_transaction(trans_id, list_of_user_ids, parent_group, database_name): this registers a grouped transaction for the users concerned, and creates a local shared database accessible to these users;

• create_shared_database(list_of_user_ids, database_name): creates a local shared database; implicitly called when a shared transaction is initiated;

• commit_shared_transaction(): commits the shared transaction, and deletes the private dataspaces; a commit takes place after design agreement between the users in the transaction;
• `abort_shared_transaction()`: abort the transaction, thereby rejecting all changes;

• `display_shared_data_pool(trans_id)`: displays the data pool of the transaction;

• `display_shared_members(trans_id)`: displays the ids of the users in the group;

• `notify_shared_members(message)`: sends message to all the group members;

• `join_shared_transaction(trans_id)`: allows an authorized user to join a shared transaction;

• `quit_shared_transaction(trans_id)`: allows an user to quit a shared transaction.

4.5.11 Transaction Checkpointing and Object Registration

Collaborative engineering implies that concurrent transactions must be able to interact in the middle of their execution. This is called transaction visibility. Visibility refers to the ability of one transaction to see the effects of another concurrently executing transaction. In traditional database systems, transactions are considered atomic, i.e., they either commit all the changes as a unit or not commit any of the changes at all. The enforcement of atomicity follows from the criteria of serializability, which attempts to order the sequence of database reads and writes. Long-lived cooperative transactions may need to share object changes prior to commit as well as access objects modified by other active transactions. For dynamic CAD transactions, atomicity therefore compromises collaboration since one transaction may not communicate its results in part to others until it commits after what may be a very long session.

In order to alleviate this problem, the concepts of transaction checkpointing and object registration [15] were introduced. Checkpointing a transaction at any point in the transaction commits all the changes made up to that point irrespective of whether the rest of the transaction aborts at a later time. This helps by saving changes made during long transactions instead of waiting for it to complete.

Object registration enables a transaction which has acquired a write or exclusive lock on object (making it inaccessible to others) to make its latest state, at any point in the transaction, visible to others without having to commit the transaction or release the locks on the object. Object registration allows specific objects from a transaction pool to be committed to the database, unlike a commit operation where the entire data pool is committed, and all locks are released. Even if the transaction is eventually aborted, the changes to registered objects are saved. Object registration sends appropriate notification to all affected users. This is particularly useful for transaction groups which are actively sharing specific intermediate results with each other without releasing the locks on the objects. It also enables a "snapshot" reader of the object to remain up to date without acquiring a restrictive lock on it.

The operators required for the above are as follows:

• `checkpoint_transaction(trans_id)`: checkpoints a transaction at any stage; object updates, if any, result in update notification;
- `register_entity(entity_id)` registers a given entity from the current transaction buffer pool to the shared database; this is accompanied by update notification to all dependents.

4.5.12 Concurrency Management and Data Consistency

One of the key requirements of a collaborative CAD environment is a flexible concurrency control mechanism that allows a high degree of parallelism and data sharing between concurrently executing transactions and maintains database correctness. Several measures for promoting greater concurrency and interaction have already been discussed, viz, versioning, flexible and communicative locking, conflict handling, transaction grouping and sharing, transaction checkpointing, object registration and semantics-based conflict resolution. The question is, do these mechanisms preserve the "correctness" of the database?

Most of the above schemes clearly violate the principle of atomicity of transactions as a measure for maintaining database consistency. There are a few approaches for ensuring database consistency despite greater concurrency. These include: application specific or semantics-based concurrency control and version management with encapsulation of cooperative design operations in "local" dataspaces.

Semantics-based concurrency control

OODBMS type systems and operations incorporate a great deal of application semantics which, along with intelligent programming, may help to define the notion of data "correctness" beyond the limited criteria set by a sequence of ordered read and writes. Most object-oriented operations might interact in ways that are at a much higher level and quite different from traditional reads and writes. This is called type-specific concurrency control. The consistency checking protocol for concurrently executing transactions is embedded in the type being operated on.

We have already considered an example earlier (Figure 4-6) where non-serializable schedule does not violate database correctness because of the semantics of the operations involved. As another illustrative example, consider objects of a class `Queue` which defines operations `Enqueue()` and `Dequeue()`[35] (Figure 4-20). If `Q` is an instance of `Queue`, then `Enqueue(Q, x)` places `x` on the input end of the queue, and `Dequeue(Q, y)` removes an element `y` from the output end of `Q`. `Enqueue()` and `Dequeue()` would be viewed by traditional systems as write operations and therefore require transactions to acquire write locks on `Q` to execute either. Thus if one transaction issues an `Enqueue()`, and another issues a `Dequeue()` on `Q`, one must wait for the other to complete. However, as long as the queue is non-empty, it may be observed that it is perfectly possible for the two transactions to proceed simultaneously without interference. Taking the semantics of the types and operations into account allows us to achieve greater concurrency than traditional models would allow. On the other hand, the same framework would serialize input and output operations on a `Stack` object because the semantics of the stack data structure necessitates it - one does not need a serializing system to enforce serializability when it is actually
required.

Object-oriented databases are ideal for this approach since they capture the higher-level semantics of classes, and class-specific operations. Several studies have been conducted to support concurrency on the basis of type-specific data and operation semantics [31, 46, 5, 39] in object oriented databases.

**Encapsulation of collaborative work**

As emphasized throughout this chapter, collaborative work such as done in transaction groups and shared transactions always take place in local databases derived from the global (or parent) databases. In such collaborative work, consistency conditions are normally relaxed to allow greater interaction, hence design objects may pass through several inconsistent states before the design is finalized. Hence, users who are not a part of these units of collaboration should be shielded from such transient data changes. This is achieved by allowing collaborative work between selected users to proceed in local shared databases. When the design is finalized in the design group, a stable design (agreed to by all members) may be committed to the global (or parent level) database, where they become visible to others outside the group. In addition a new version of the object is created due to the update, so it is always possible to back up to a previous state. Thus in the global database objects are versioned from one stable design state to another, with intermediate and inconsistent states being buried in local databases.

**4.6 Summary**

In this chapter, we have presented a model for collaborative CAD and discussed the transaction management framework for facilitating coordination and concurrency using OODBMS for such work. Key issues that have been emphasized include communication between designers, sub-division of transactions into sub-transactions for better management, transaction grouping and local, shared partitioned databases for active data sharing and experimentation, version management for parallelism and documentation of design evolution, greater flexibility in concurrency management by intelligent programming and use of type systems and application semantics to ensure data consistency. The limitations of traditional DBMS for such collaborative environments have been enunciated.

The next chapter discusses implementation of some of the transaction management features using ONTOS, a commercial OODBMS.
**Figure 4-20:** Example of type-based concurrency management
Chapter 5

Implementation Issues

5.1 Introduction

This chapter describes the implementation of some of the features of the proposed transaction management framework using a commercial OODBMS, ONTOS on a network on Sun3 machines running Sun's UNIX, SunOS 4.1. ONTOS uses C++ as the front end data definition and manipulation language. The various features of ONTOS have been described in detail in Chapter 3.

ONTOS was chosen as the OODBMS for our implementation because (a) it is C++-based, so that it could be easily interfaced to other modules of DICE code which are also being developed in C++; (b) it was the only C++-based systems available commercially at the time we started implementation; and (c) ONTOS was made available to us for educational purposes.

However, like all other commercial OODBMS, ONTOS is still evolving and lacks most of the features proposed in our model. These features were implemented over the existing ONTOS framework. However, since ONTOS is so restricted in some of its transaction functionalities, and the source code was not available to us for enhancements, some of the basic limitations still remain, and at best can only be simulated over the ONTOS environment.

The following sections describe the transaction management features that are being implemented over ONTOS.

5.2 A Dynamic Transaction Framework

ONTOS transactions are defined statically (at compile time) in C++ code, and are not changeable at run-time. This provides a poor environment for collaborative work, where user actions are interactive and are largely defined dynamically. Also, due to the static nature of the C++ language, new data structures (e.g., new class definitions) cannot be added at run-time, nor can existing data structures be modified. To alleviate this problem, a dynamic linker has been developed which links in C++ code defined at run-time into the running application without having to stop it. It also allows the definition of new classes at run-time. This gives the user the flexibility
required to incrementally define and develop his/her design application.

5.2.1 Dynamic Linking

*Dynamic linking* refers to the ability to add new code to a running program - new code that can then be accessed and executed from within the old code. Variations of dynamic and incremental linkers for C and C++ have been implemented and is a well understood procedure [13, 55, 56].

The procedure for dynamic linking is as follows (Figure 5-1): the user develops a body of *new* C++ code (that is to be incorporated into the running application) in a file, and compiles it to its object code. The system then incrementally links in only this new object code using the dynamic linker provided on Sun3, executes the new piece of code, and returns control to the main running application.

5.2.2 Implementation details

The internal workings of the dynamic linker are as follows:

1. First, the main object file (for the running application) is located in memory;

2. Next, the shared library symbol tables are loaded into memory. The libraries, and the order in which they are loaded is controlled by the global variable "LIBRARIES". (The UNIX utility bfin 1ld may be used to find out which shared library the code depends on). Loading the libraries and the main object code builds the stack of symbols. During lookup, a symbol is first looked for in the loaded object files (most recently loaded first), then the main program, and finally in the shared libraries;

3. The new piece of object code is now loaded. The text and data segments of the object file are loaded into memory followed by the symbol table associated with the new code;

4. *Symbol table assimilation* is done next. All new global variables introduced in the new code (e.g., a new class definition, or a new function) are added to the existing symbol table of the main application, so that these variables may be accessed by new pieces of code added dynamically later on. This allows the application to grow incrementally.

5. In the case of *symbol redefinition* (e.g., a change in the definition of a class), the new (redefined) symbol is installed in the place of the old. The implications of this redefinition on existing instantiations of the data structure is a complicated research issue and is still open to investigation. Most systems that do allow dynamic linking (e.g., Saber-C [55] and Saber-C++ [56]) do not allow redefinition of existing data structures (the results are unpredictable if done).

In our implementation, we provide for a wide range of redefinition capabilities because of its usefulness for *dynamic schema evolution*. We will illustrate these with several examples in the following section.
Figure 5-1: Dynamic linking
5.2.3 Examples of Dynamic Linking

Here, we present various example cases of dynamic linking using the dynamic linker. All code is in C++.

1. Adding a new function to the application.
   The following piece of code shows the definition and incorporation of a new function into the application at run time.

   // file test1.c
   # include <stdio.h>
   # include <string.h>
   # include "inc.h" // a standard header file for dynamic linking

   // introduce new function here

   int add(int a, int b)
   {
       return (a+b);
   }

   test1()
   {
       printf("new function add() installed \n");
   }

   The above code, when compiled and linked, incorporates the definition of the function add() incrementally in the application, so that it may be called later in another piece of dynamically linked code as illustrated below.

2. Invoking the new function from another piece of dynamically linked code.
   The following incremental code invokes the function add() which was added dynamically at run-time.

   // file test2.c
   # include <stdio.h>
   # include <string.h>
   # include "inc.h"

   extern int add(int a, int b); // declare the function as
                          // an extern

   test2()
{  
  // call the new function add() from here
  int res = add(1,2);
  printf("RES: %d \n", res); // prints 3
}

3. Modifying an existing function at run-time.
The following code redefines the function add(), and invokes it to show that it has been changed internally.

// file test3.c
#include <stdio.h>
#include <string.h>
#include "inc.h"

// redefine the function
int add(int a, int b)
{
  return (a+b+10);
}

test3()
{
  printf("Function add() redefined \n");
  printf("TEST: %d \n", add(10,10)); // prints 30 of course!
}

4. Adding a new data structure (e.g., a new class) at run time.
Assume we want to install the following class A (defined in file A.h) in the application at run time:

// file A.h
#ifndef A_H
#define A_H
class A
{
public:
  int a_1;
  A()
  void print_slot();

111
void set_a_1(int);
}

void A::print_slot()
{
    printf("a_1: %d \n", this->a_1);
}

void A::set_a_1(int x)
{
    a_1 = x;
}
#endif

The following piece of code installs the class A and instances it into a1.

// file test4.c
#include <stdio.h>
#include <string.h>
#include "inc.h"

// define class A
#include "A.h"
A* a1 = new A;
test1()
{
    a1->a_1 = 100;
    printf("Class A installed; instance a1 created \n");
}

5. Change the data structure of a class.
Let us now redefine the class A as in the following code. We will drop attribute a_1 (type int), and add a new variable a_2 (of type char*). The existing instance a1 is now transformed into an instance of the new template A. The functions print_slot() and set_a_1() therefore lose their meaning, since attribute a_1 does not exist. However, a1 dynamically acquires the new attribute a_2, and may store values in this slot. Also, if we create a new instance a2 from the new class definition, an invocation of print_slot() on a2 returns zero (since slot a_1 does not exist) instead of crashing the application.
// file test5.c
#include <stdio.h>
#include <string.h>
#include "inc.h"

// redefine class A

class A
{
  public:

    char* a_2; // a new slot added, slot a_1 dropped
    A()
    {
      void print_slot();
      void set_a_1(int);
    }

    extern A* a1; // obviously a1 gets "recast" to the new type and
    // acquires all kinds of new slots from the new
    // class A;

    A* a2 = new A; // new instance from the new template

  test5()
  {
    a2->print_slot(); // prints 0, since a2 does not know about a_1

    a1->a_2 = new char[strlen("100")+1];
    strcpy(a1->a_2, "100"); // ok, a1 acquires new slot a_2
  }

  In the above example, a redefinition of class A overwrote the previous definition
  leading to problems of instance m. . xation. A better approach to handling such
  schema evolution is to create a new version of class A (as discussed in Chapter
  4) and preserve the structure and behavior of existing instances in the extension
  of that version of the class.

  6. Dynamic definition of ONTOS transactions. Using the dynamic linker,
  ONTOS transactions may be defined and executed dynamically instead of hav-
  ing to define them at compile time. The following is a sample ONTOS trans-
  action that was defined and executed dynamically.

  trans()
{  
    printf("Starting first transaction trans1 \n");  
    // create a transaction  
    Set* trans_set = my_OC_transactionStart("trans1");  
    printf("\nTransaction set %s created \n", trans_set->Name());  
    root* some_obj = (root*)my_OC_lookup("obj", trans_set, ReadLock);  
    if (some_obj == NULL)  
    {  
        // create a new object in the database and post it  
        printf("Creating new object \n");  
        some_obj = new root("obj");  
        printf("\nObject %s created \n", some_obj->Name());  
        some_obj = (root*)my_OC_lookup("obj", trans_set, ReadLock);  
        some_obj->putObject();  
    }  
    // end of transaction  
    my_OC_transactionCommit(trans_set);  
    printf("First transaction ended \n");  
}

Eventually, the dynamic linker will be replaced by a true C++ interpreter such as Saber-C++, which is currently in its beta release. A fourth generation interpretive language (4GL) is currently under development at Ontologic, Inc.

5.3 Client, Object and Transaction Record

ONTOS does not make records of clients using the database available to the application, nor does it maintain any object ownership and timestamp records that may be accessed from the program. The present framework maintains persistent object representations of all users (in instances of class User) of the database and their activities (namely, time and id of transactions initiated, objects created and modified, etc.). Similarly, every object maintains a record of its ownership, time of creation and modification, and a dependency table of all clients who have accessed it. This is achieved by deriving all user defined classes from a base class called root, which provides the attributes and methods to keep a record of all information. Each object also maintains a History_dictionary object which records the changes made to the object with timestamps. For the purpose of transaction logging, each transaction instantiates a persistent Transaction object which stores the id of the transaction, its initiator, the data buffer pool affected by the transaction, and the changes made to existing objects during the transaction. The code for the classes User, root, History_dictionary and Transaction is included in the Appendix. The hierarchy of all defined base classes is shown in Figure 5-2.

ONTOS does not provide any form of database security, which is also a hazard,
Figure 5-2: Hierarchy of defined base classes
since unauthorized users may access certain data. The ownership record will eventually be expanded to maintain some form of authorization information whereby only certain users may access and share objects and unprivileged access attempts will trigger warnings to users.

5.4 Communication Protocols

ONTOS does not support any form of inter-client communication so essential for cooperative work. Mechanisms have been implemented for client notification in the event of lock requests, attainments, releases and conflicts. Since ONTOS does not provide a record of the lock table or that of the clients using the database, persistent object lock tables in the form database objects have been implemented for recording client information and their activities (transactions and updates to objects). Users on remote workstations may also exchange messages with each other. Currently, the messages are sent using UNIX communication tools; an X Window based user interface for providing pop-up message windows and mailboxes is currently under development.

5.5 Update Notification

ONTOS does not provide any object update notification schemes. In the present framework, every object maintains a dependency table (in the form of a Dependency dictionary object) of clients that have accessed the object anytime. When the object is changed and committed, update notifications concerning the nature of the update, the client id and the time of the update are sent to all affected users, and are logged in a persistent record. In addition, an user may also explicitly notify other dependents of any changes in an object without actually committing the object to the database. The source code for the Dependency_dictionary object is provided in the Appendix.

5.6 Interactive Lock Management

ONTOS does not provide information regarding the lock status of an object. In our implementation, every object maintains a persistent lock dictionary (as a Lock_dictionary object) which stores the ids of the clients holding locks on the object and the types of locks held. Before requesting a lock on an object, an user may check the lock status of the object. In the event of lock attainments, releases, queuing, denials and conflicts, messages are sent to all users concerned regarding the situation.

5.7 Transaction Conflict Management

ONTOS generally tends to abort transactions (with embarrassing segmentation violations in the application) which are in conflict. While the basic handling of transactions
could not be changed, better exception handling facilities have been provided so that users may gracefully abort their transactions, or do something else to remedy the situation rather than have their applications crash. When ONTOS detects a potential conflict between transactions, the exceptions raised notifies users of the conflict. The users may then abort their transaction immediately, wait or retry later again, or may perform other tasks to handle the situation alternatively. As emphasized earlier, effective communication between users will help to avert system-dictated aborts, so that the clients may resolve the situation in a most semantically reasonable manner.

The following code illustrates a typical exception handling scheme in the application:

```c
ExceptionHandler trans_conflict("WaitException", user_resolve);

OC_transactionCommit(RWConflict, OC_notifyConflict);
if (trans_conflict.Occurs()) // user_resolve function gets called
  // here
{
  printf("Conflict taken care of \n");
  OC_transactionCommit();
}
```

The above creates the exception handler trans_conflict which detects the system defined exception WaitException and invokes the user defined conflict resolution function user_resolve() upon transaction conflicts. If the conflict does occur, user_resolve is executed which sends notification to conflicting clients allows various options rather than arbitrarily aborting the transaction.

## 5.8 Version Management

Currently ONTOS does not provide version management of objects. In the present framework, a generic persistent versionable class (version_obj) has been provided such that all its instances (or instances of classes derived from the generic class) are automatically versioned. Changes to such objects result in the creation of a new version object which is added to the object's version set (in the form of a Version_set object). Each versionable object maintains a record of its version count, its creator, a comment from the designer, a timestamp, and predecessor and successor links, thereby forming a version hierarchy. Object versions are uniquely named so as to indicate the version number and the version it was derived from (the first instance is called the root version). Users may list all versions, explicitly request any particular or latest version, and delete any version. Deletion of a version object from the hierarchy leads to its successor versions being concatenated to the successor links of its predecessor object. Since different users may define their own versions from the same object independently, branching or parallel versions may be created.
Currently, version management facilities have been implemented only for instances of classes. Since C++ is a static language, and does not allow redefinition of data types at run-time, it is considerably more difficult to implement version management of classes. However, this may now be possible using the using the dynamic linker (the original structure of a class A in the static code may be redefined in the dynamically linked code), and is presently under consideration. This may be implemented by defining what may be called a meta class, a class each of whose instances store the extensions (all the instances) of a class. When the class structure is changed, a new meta class is created which stores the instances of the new version of the class. The structure of meta class is as follows:

class meta class: public root
{
    char* classname; // the class whose instances are stored
    char* class_version; // the version of the class
    Set* class_extension; // all the instances of the given
    // class version
    // other structure and behavior
};

Versions of class hierarchies will have to be deferred until there is a better understanding of how to deal with schema changes in large databases.

5.9 Version Configurations

This refers to versions of composite objects, collections and aggregates, or in other words, management of configurations of objects, rather than that of a discrete object.

5.9.1 Versions of Composite Objects

C++ (and ONTOS) lacks the notion of a composite object, and a model is currently being developed over C++ [41, 49]. Since the model is not a part of the language, it is being implemented programmatically, i.e. by simulating the behavior of composite objects through application-specific code. Issues concerning the complexity of version management of composite objects have been discussed in Chapter 4. Versioning schemes would therefore have to be type and application specific, i.e., each object would have its own version management scheme depending on the nature of the changes, the depth of the component at which the changes are made, etc. Refer for example, Figures 3-2, 4-13 and 4-14, Chapter 4.

5.9.2 Versions of Aggregates

ONTOS provides standard collection classes such as Set, List, Array, Dictionary, etc.; however it does not provide any version management schemes for such classes.
We would like all user defined collections have all the properties of these standard classes, as well as those of root (for record keeping) and version_obj (for version management). Ideally, we would therefore use multiple inheritance to derive our versionable collection class, my_Set, let's say from these base classes, as shown in Figure 5-3. However, ONTOS does not yet support multiple inheritance for persistent classes, hence this scheme could not be used. Hence, my_Set was derived from Set and all the functionalities of root and version_obj were duplicated in it. Currently, versions of collections are maintained at the component level if the components are versionable. If any member of a collection is changed, that component is versioned, while the collection is not (Figure 5-4). If the components themselves are not versionable, then we must create a new version of the collection object itself. However, for components that are unchanged in the new version, pointers are maintained to corresponding components in the parent version. (Figure 5-5).

5.10 Database Partitions

ONTOS does not support database partitions into private and local shared areas. Also, an application can access only one database in a particular session. Since it is not possible to discretized the database at a physical level, database partitioning may only be simulated inside the application. This is done by defining a class Database which serves a repository of selected objects and provides access to selected users. Each Database object has pointers to its parent database (from which its data is derived and to which it commits its changes) and children databases. The partial structure of the class is as follows:

class Database: public root
{
  char* database_name; // the name of the partition
  List* database_users; // the users who may access the objects in
  // the database
  Set* database_objects; // the objects in the partition
  // other structure and behavior
};

The creation of such Database objects incurs considerable overhead in record keeping and processing because the number of objects in the repository may be large and all the regular functionalities of a database have to implemented for the class. Besides, Database objects being normal objects in the ONTOS database themselves are manipulated (e.g., activated, accessed, locked, updated, committed, etc.) by conventional ONTOS methods which are not amenable for shared transaction processing. For example, updating two different objects in repository by two different users would require a write lock by each on the database object itself! Update conflicts in such cases may not be easy to resolve because a large number of repository objects may have been updated for every Database commit.
Figure 5-3: Desired setup for versionable collection classes
Figure 5-4: Version management of collections with versionable components
Collection object

$\text{change}$

$=$

Collection object version 1  +  Collection object version 2

Figure 5-5: Version management of collections with non-versionable components
5.11 Implementation Problems Using ONTOS

Due to the conservative and restricted functionalities of the ONTOS framework, several advanced features (such as transaction nesting, grouping, flexible locking protocol, etc.) are impossible or otherwise difficult to implement. One of the problems already mentioned is the lack of multiple inheritance facilities. Secondly, ONTOS strictly enforces two-phase locking whereby it is not possible to release locks on objects (after they have been used) until the end of the transaction. This causes problems with record keeping itself. For example, assume two users are simultaneously trying to acquire read locks on the same object in two different transactions. Such requests do not conflict, and therefore will be granted by the system. However, this requires the Lock_dictionary of the object has to be updated from within both transactions to record the locks acquired by both the users. Therefore the transactions will be in conflict even though the lock requests themselves are not, the problem being that one transaction cannot release the write lock on Lock_dictionary after the lock acquisition is recorded. The solution to this problem is to create two temporary versions of Lock_dictionary to store the two lock acquisitions and then merge them together (Figure 5-6. The Lock_dictionary is not intended to be versionable and creation of such versions is unnatural, since logically there should only be one record of the lock history (just as a person cannot have multiple versions of the same bank account).

The other problems in using ONTOS concern its fragility. It tends to crash over the network frequently. It also causes segmentation faults in applications that run perfectly with C++ alone. ONTOS also does not allow a database to be opened, closed and reopened in the same session.

5.12 Summary

This chapter has presented implementation details of some of the features of our transaction management model. The flexibility and scope of implementation is restricted by the limitations of the database management system being used. For example, ONTOS enforces serializability and two-phase locking for concurrency control and does not allow database partitioning or shared databases and nesting of transactions. However, the framework implemented significantly alleviates the problems of communication, coordination and record keeping and version management enables greater degree of concurrency between transactions. More advanced features and several other OODBMS are currently under investigation. These and other research directions are discussed in the next chapter.
Problem of simultaneous update of X_Lock_dictionary

<table>
<thead>
<tr>
<th>Transaction 1</th>
<th>Database Response</th>
<th>Transaction 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read_lock (X)</td>
<td>Granted</td>
<td>Read_lock (X)</td>
</tr>
<tr>
<td>=&gt; Write (X_Lock_dictionary)</td>
<td>Conflict</td>
<td>=&gt; Write (X_Lock_dictionary)</td>
</tr>
</tbody>
</table>

Work-around

<table>
<thead>
<tr>
<th>Transaction 1</th>
<th>Database Response</th>
<th>Transaction 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read_lock (X)</td>
<td>Granted</td>
<td>Read_lock (X)</td>
</tr>
<tr>
<td>=&gt; Create temporary version X_Lock_dictionary_1 storing acquisition by T1</td>
<td>No conflict</td>
<td>=&gt; Create temporary version X_Lock_dictionary_2 storing acquisition by T2</td>
</tr>
</tbody>
</table>

Merge into X_Lock_dictionary

Figure 5-6: Problem of recording simultaneous lock acquisitions on an object
Chapter 6

Conclusion

6.1 Summary

This study has presented a framework for facilitating and coordinating collaborative engineering design using object-oriented databases. It forms an integral component of the ongoing DICE effort, where the initiative has been the development of an integrated environment for planning, design, and control of engineering CAD and manufacturing processes. The key features of our transaction management model that have been emphasized include:

- a dynamic and interactive transaction framework for interpretive definition of transactions at run time and incremental application development;

- good inter-client communication and update notification facilities which enable interaction between designers and keep them aware of each others’ work;

- flexible and communicative locking protocols which alleviate the problems of having to wait indefinitely for acquiring locks on objects;

- version management facilities to enhance the parallelism of design activities and to keep track of the evolution of design;

- rigorous constraint management by embedding constraints on design data in the database, and flagging constraint violations by designers;

- robust conflict handling facilities for detecting lock, commit and design conflicts between users and notifying them of the conflict so that the problem is resolved in a semantically reasonable manner instead of aborting one or more transactions arbitrarily;

- hierarchical design task discretization in the form of nested transactions representing specific design activities;

- transaction grouping between several cooperating users for local data sharing in a specific area of design;

125
• *shared transactions* between two or more users for more active data sharing and interaction without having to cross transaction boundaries;

• *database partitioning* in the case of nested, grouped and shared transactions into *local shared areas*, which are accessible only to collaborating users and are inaccessible to others;

• data *check-out* and *check-in* facilities between database partitions which allow users to work on design objects for *long durations* in their private or local shared databases without restricting the access privileges of others; only stable design data is checked back in which creates a new version of the existing data;

• *flexible (non-serializable) concurrency control* in collaborative grouped or shared transactions by *exploitation of application semantics*, versioning and extensive communication between users;

• enforcement of *data consistency* by *encapsulation* of non-serialized data sharing in local databases and localized specification of *design semantics* and *constraints* to ensure validity of data;

• *graphical* tools to visualize design versions, transaction nesting and grouping, etc.

The advantages of using such an environment over traditional systems for CAD applications have been enunciated.

The current state of this evolving technology has been described along with difficulties encountered in its formulation and implementation. Transaction frameworks for OODBMS which are built on semantics-based multilevel concurrency protocols are still in their infancy. More pressing implementation problems concern the development of a flexible transaction management environment with emphasis on effective coordination and communication between designers, both of which are conspicuously absent in most current CAD environments. A few OODBMS research prototypes which support most of the advanced features discussed in this paper are under development, the most notable being Observer/ENCORE database system [18].

A comprehensive description of the principal functionalities of currently available OODBMS, and their relative strengths and weaknesses for engineering design database management have been studied. Their merits over RDBMS have been emphasized, namely in the power of complex data modeling and abstraction, expressiveness, and in the handling of flexible CAD transactions. Object-oriented database environments are still evolving and show considerable promise for being the standard platform for modeling, storage and manipulation of engineering information in the foreseeable future.

### 6.2 Current and Future Work

A complete implementation of the proposed model using the tools available at this stage is an onerous task. While the notion of serializability is an established and
proven concept, theoretical issues concerning concurrency management (such as recovery, rollback, etc.) in non-serializing systems are yet unclear. The semantics-based correctness criteria also assume intelligent and cautious programming, since the system no longer ensures consistency of data; hence such systems may not be amenable to naive users. It also requires a strong constraint generation and checking system to detect anomalous designs (ongoing work in this discipline is described in [42]). The drawback with this approach is that concurrency control protocols become highly application (or type) specific and it is difficult to develop a generalized system (like serializability-based ones) which will ensure data consistency for all applications (and types).

Issues concerning the implementation of our transaction management functionalities using the ONTOS commercial OODBMS has been described. Due to the restricted functionalities of ONTOS, several advanced features (such as transaction grouping, flexible locking protocols and database partitioning) are impossible or otherwise difficult to implement. The following features have been implemented over the ONTOS framework:

- a *dynamic linker for C++* for dynamic definition of ONTOS transactions, schema evolution and incremental design development;

- facilities for *maintenance of client, data and transaction records* such as object ownership and timestamp, update histories, locking and dependency dictionaries, transaction buffer pools, etc.;

- *version management* of object instances and collections, with facilities for version links and flagging differences between versions;

- *inter-communication facilities* for interaction between users and notification of database events;

- *communicative locking facilities* for informing users about lock status of objects, lock attainments, releases, denials, etc.;

- *update notification facilities* which notify dependent users of changes to data accessed;

- *robust conflict handling facilities* which detect transaction commit conflicts and raises exceptions to notify conflicting users instead of letting ONTOS abort transactions arbitrarily;

- *database partitioning* has been implemented programmatically, since it was not possible to do physical partitioning of the database.

Transaction nesting and grouping protocols are currently under development at Ontologic, and will be available soon. Considerable work is also necessitated in the area of graphical user interfaces (for depicting version hierarchies, client-object dependencies, lock-queue graphs, etc.), version management of classes and composite
objects, hierarchical database partitioning, etc. Integration of these diverse sub-components into one functional and reliable design environment is the real challenge for the future.
Appendix A

Transaction Management Classes

Here, we include the C++ source code for various classes that were used to implement our transaction management system.

A.1 Class root

The class root is the base class from which all application defined classes are derived. It maintains all the data structures for keeping record of the creator, timestamping, history of modifications to the object, the lock and dependency dictionaries of the object, etc. The C++ code for the class declaration is as follows:

```cpp
class User; // clients using the database

class root: public Object
{
public:
    char owner[30]; // name of the owner
    time_t time_created; // time of creation of object
    char converted_time_created[30];
    time_t date_modified; // time this object was modified last
    char converted_date_modified[30];
    char person_modified[30]; // userid of person who
                           // last modified this object

    Lock_dictionary* lock_dictionary; // the dictionary in
                                        // which locks reside
    Dependancy_dictionary* depend_dictionary;
    // dictionary containing ids of all clients
    // who have accessed the object

    History_dictionary* history_dictionary;
    // records the history of changes
```
root(APL* theAPL); // APL constructor

root(char* Name); // new instance constructor
virtual void update_modify();

virtual void add_locker(char* u_id, char* locktype);
virtual void add_dependant(char* u_id);

virtual void remove_locker(char* u_id);
virtual void remove_dependant(char* u_id);

virtual void inform_all_lockers(char* new_uid, char* locktype);
virtual void inform_all_dependants();

virtual void display_all_lockers();
virtual void display_all_dependants();

virtual void add_record(time_t , User* );
virtual void show_history();

virtual void display_data();

virtual void deleteObject(Boolean deallocate = FALSE);

void Destroy(Boolean aborted = FALSE)
{
    Object::Destroy(aborted);
}
};

A.2 Class User

The class User represents all users of the ONTOS database, and is used for maintaining dependency, history and locking records. It records the user's id, the type of lock requested on an object, and the action taken.

class User: public Object
{
  public:
    char* userid;
    char* locktype;
    char access_time[30];
char* action;

User(APL* theAPL);
User(char* c_id, char* c_lock, char* the_action = 0,
    char* name = 0);
void display_data();
void set_action(char* action);
// records action by user, eg, read, update, etc.
void Destroy(Boolean aborted = FALSE);
}

A.3 Class Transaction

Instances of the class Transaction represent individual transactions initiated by various clients. It keeps a record of the initiator, the time at which the transaction was initiated, the data pool accessed, and the transaction status.

class Transaction: public Object
{
    public:
    char* id; // id of the transaction
    char* client_name; // client who initiated
    // transaction
    time_t time_initiated; // time of initiation
    // of transaction
    char converted_time_initiated[30];
    Object_set* data_pool; // the names of the objects
    // accessed
    char* status; // abort or commit
    Transaction(APL* theAPL);
    Transaction(char* trans_id, char* username);

    void add_object(char* object_name);
    void delete_object(char* object_name);

    void display_data_pool();
    void display_data(); // displays info on
    // client_name, timestamp, status
    void Destroy(Boolean aborted = FALSE)
    {
        Object::Destroy(aborted);
        delete id;
        delete client_name;
        delete status;
    }
A.4 Class Dependency_dictionary

Every object maintains a Dependency_dictionary, which contains the ids of all users who have accessed the object, either for reading or update. This enables the system to send update notification messages to users upon change in the object.

class Dependency_dictionary: public Dictionary
{
public:
  char* object_name;  // name of the object for which this dict exists
  Dependency_dictionary(APL* theAPL);
  Dependency_dictionary(Type* indexspec, Type* memberspec,
    Boolean isordered = 0, Boolean hasDuplicates=0,
    char* name= 0, char* obname = 0, Object* where = 0,
    Clustering howNear=defaultClustering);

  void display_all_dependents();
  int remove_dependent(char* u_id);
  int add_dependent(char* u_id);
  void inform_all_dependents();
  User* get_particular_client(char* );
  void Destroy(Boolean aborted = FALSE);
};

A.5 Class History_dictionary

Every object maintains a History_dictionary to keep a timestamped record of the history of changes made to the object, and of the nature of those changes. This also helps to keep track of the evolution of design.

class History_dictionary: public Dictionary
{
public:
  char* object_name;  // the object for which this dictionary exists
  History_dictionary(APL* theAPL);
  History_dictionary(Type* indexspec, Type* memberspec,
    Boolean isordered = 1, Boolean hasDuplicates=0,
    char* name= 0, char* obname = 0, Object* where = 0,
Clustering howNear=defaultClustering);

void add_record(time_t the_time, client* the_client);
void show_history(); // shows history since certain time
void Destroy(Boolean aborted = FALSE);
};

A.6 Class version_obj

The class version_obj is the superclass of all application defined versionable classes. It keeps track of version number, timestamp, creator, preceeding and succeeding versions, and the version set of the object. The following is the source code for this class:

class Version_set; //forward declaration

class version_obj: public root
{
public:
    int version_number;
    char* version_set_name[100];
    version_obj* derived_from; // the preceeding version
    Set* derived_versions; // all versions derived
                        // from this version
    version_obj(APL* theAPL);
    version_obj(char* Name);
    version_obj(char* Name, int ver_no, version_obj* der_from);

    Set* get_derived_versions();
    version_obj* get_preceeding_version();
    version_obj* derive_new_version();
    void display_data();
    void derive_modify();
    Version_set* get_version_set();
    void insert_derived_version(version_obj* new_obj);
    void remove_derived_version(version_obj* poor_obj);
    void show_derived_versions();
};

A.7 Class Version_set

Version_set holds all the versions of a given versionable object. Versions in the version set may accessed, deleted, displayed, etc.
class Version_set: public Dictionary
{
    public:
    Version_set(APL* theAPL);
    Version_set(Type* indexspec, Type* memberspec,
        Boolean isordered = 0,
        Boolean hasDuplicates=0, char* name= 0,
        Object* where = 0,
        Clustering howNear=defaultClustering);
    int last_version_no;
    int delete_version(int ver_no);
    int get_number_of_versions();
    void display_all_versions();
    version_obj* get_particular_version(int ver_no);
};

A.8 Class Lock_dictionary

Every object also maintains a Lock_dictionary which records the various types locks held on the object at a given time, and the users holding those locks. This facilitates communicative locking.

class Lock_dictionary: public Dictionary
{
    public:
    Lock_dictionary(APL* theAPL);
    Lock_dictionary(Type* indexspec, Type* memberspec,
        Boolean isordered = 0,
        Boolean hasDuplicates=0, char* name= 0,
        Object* where = 0,
        Clustering howNear=defaultClustering);

    void display_all_lockers();

    void remove_locker(char* u_id, char* objname);
    void add_locker(char* u_id, char* l_type);
    void inform_all_lockers(char* u_id, char* ltype);
    User* get_particular_client(char* u_id);
    char* check_conflict();
    void Destroy(Boolean aborted= FALSE);
};
A.9 Versionable Collection Objects

Collection objects in ONTOS (such as Set, Dictionary, Array, etc.) have been redesigned to be versionable. However, as discussed in Chapter 5 (Figure 5-3) since ONTOS does not permit multiple inheritance, these new classes could not be multiply inherited from version_obj and the primitive ONTOS collection class. Hence all the functionalities of root and version_obj had to be reimplemented inside the new class. The following is the code for the class my_Set, a versionable Set class.

class my_Set: public Set
{
  public:
    char owner[30];  // name of the owner
    time_t time_created;  // time of creation of object
    char converted_time_created[30];
    time_t date_modified;  // time this object was modified last
    char converted_date_modified[30];
    char person_modified[30];  // userid of person who
    // last modified this object
    Lock_dictionary* lock_dictionary;  // dictionary in
    // which locks reside

    int version_number;
    char version_set_name[100];
    my_Set* derived_from;  // the preceding version
    my_Set* derived_versions;  // all versions derived from this version
    Set* get_derived_versions();
    my_Set* get_preceeding_version();
    my_Set* derive_new_version();
    void display_data();
    void derive_modify();
    Version_set* get_version_set();
    void insert_derived_version(version_obj* new_obj);
    void remove_derived_version(version_obj* poor_obj);
    void show_derived_versions();

    my_Set(APL* theAPL);
    my_Set(Type* memberSpec, char* Name = 0,
            Object* where = 0,
            Clustering hownear = defaultClustering);
    my_Set(Set& anotherSet);

    virtual void update_modify();

    virtual void add_locker(char* u_id, char* locktype);
}
virtual void remove_locker(char* u_id);
virtual void inform_all_lockers(char* new_uid, char* req_locktype);

virtual void display_all_lockers();
virtual void display_data();

void Destroy(Boolean aborted = FALSE)
{
    Set::Destroy(aborted);
}

};
Bibliography


137


[40] Sriram, D., personal communication, IESL, MIT.


[51] Zdonik, S. B., and Wegner, P., "Language and Methodology for Object-Oriented
Database Environments," Proceedings, Nineteenth Annual Hawaii International
Conference on System Sciences, 1986.

[52] ITASCA Distributed Object-Oriented Database Management System, Technical
Summary, Itasca Systems, Inc., MN.


[54] Programming in OPAL, Version 1.5, Servio Logic Development Corporation,
1989.


1990.