NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

ISBN 0-315-50113-0

Canada

Concurrency Control Scheme for Design Object Bases

T. Srinivasan Narayanan

A Thesis

in

The Department

of

Computer Science

Presented in Partial Fulfillment of the Requirements
of the Degree of Doctor of Philosophy at
Concordia University
Montréal, Québec, Canada

August 1989

ABSTRACT

Concurrency Control Scheme for Design Object Bases

T. Srinivasan Narayanan, Ph.D.
Concordia University, 1989

Serializability of concurrently running transactions has been used as the only correctness criterion in traditional DBMS. This has been recognized as a strict condition for some of the non-traditional transactions like design-activity. Recent research has focussed on the need to suitably modify concurrency control criterion to make it applicable to non-traditional applications. It is our thesis that enforcement of constraints defined on an application, domain or database behavior should be the goal of concurrency control schemes. These constraints define the correctness criteria and as long as these correctness criteria are satisfied it is immaterial whether the concurrent execution of transactions is serializable or not. Thus, more concurrency can be allowed without compromising the correctness of the results.

Traditional transactions also limit the sharing of information among the transactions thereby making them inapplicable to design environments. Uncontrolled sharing can create havoc. However, controlled and monitored sharing can be very useful. The runtime checking of these constraints can potentially be very expensive. However, efficient consistency enforcement mechanisms (*alerters* and *triggers*) are features of an increasing number of new database systems.

In this thesis we present a design activity model, where a *design activity* involves other *design activities* and/or *transactions*. A transaction is a set of actions; it is also a unit of *consistency* and *atomicity*. We associate two types of constraints, viz. *validation* and *visibility* constraints with each design activity; this also holds true for the traditional view of transactions. Validation

constraints specify correctness criteria and the conditions under which a design activity is deemed *complete*. They allow us to capture the application dependent and independent correctness criteria for the concurrent execution of design activities. Visibility constraints specify the conditions under which *sharing* among design activities occurs. They are used for purposes like *notification*, *dynamic updates*, and *versions*

We also suggest an optimistic validation scheme *dynamic validation scheme* for object-oriented databases. In this scheme validation constraints are verified at the time of the completion of a design activity.

Acknowledgements

I wish to express my deep gratitude and appreciation to Dr. P. Goyal and Dr. F. Sadri, whose guidance, encouragement, and advice has enlightened my way. I am especially indebted to them for many stimulating discussions and helpful suggestions.

I would like to thank Dr. V.S. Alagar, Dr. H.F. Li, and Dr. M. Okada for their encouragement. I would like to thank Dr. T. Radhakrishnan and Mr. C. Grossner for allowing me to use the Microprocessor Systems laboratory for drawing the figures.

I would like to thank Dr. H.F. Korth from University of Texas at Austin. His valuable suggestions were useful in the preparation of this final draft.

I would also like to thank my friends Mr. R.N. Prasad, Mrs. Raj Goyal, Mrs. Ramani Subramanian, and Mr. R. Subramanian for their encouragement and help.

Last, but not the least, I thank my parents, brother T.S. Raghavan, and my wife Vasumathi. Their understanding and support have been invaluable to me.

The research reported in this thesis was done under grants from FCAR and NSERC of Canada.

Table of Contents

List of Figures

Special Symbols

Notation	Represents
$o_1, o_2, \ldots$	Objects
O_i	Object Manager (OM) of o_i
$t_1, t_2, \ldots$	Transactions
T_i	Transaction Manager (TM) of t_i
$d_1, d_2, \ldots$	*Design activities*
s_i	Current state of o_i
s_{ij}	t_j's private copy of s_i during *execution*
s'_{ij}	t_j's private copy of s_i during *re—execution*
$a, b, \ldots$	Steps of a transaction or an action
$A, B, \ldots$	Actions
A, B, ...	Sets of actions
$S_1, S_2, \ldots$	States of the object base
$\wedge$	Logical and
$\exists$	quantifier there exists
$\forall$	quantifier for all
$\times$	Cross Product
$\cup$	Union
$\in$	Member of
i, j, l, m, n	Variables

Chapter 1

Introduction

Conventional databases have proved useful for highly structured applications involving simple entities that could be modeled by records. With the evolution of the object-oriented data model [Keta86, Mano86, Beee87] the scenario is changing [Ditt86]. Unstructured applications such as design that use complex objects are being developed using object-oriented databases [Atwo85, Buch86, Hard86, Rao86, Spoo86, Zdon86, Kim88, Walp88]. In addition to using object-oriented databases - simply called object bases [Alme85, Blac85, Ahls86, Ande86, Dasg86], existing data models are also being enhanced [Care86, Schw86, Ston86a, Ston86b] to accommodate the development of unstructured applications.

Design is typically an unstructured activity. Normally a designer works in parallel on multiple aspects of a problem. Thus, a design activity can be modeled as a set of design-actions,[1] whose execution follows a partial order dictated by the underlying application domain. We can visualize a design activity as a set of design-actions punctuated with pauses for thinking, organizing, etc. Design is also characterized by cooperation among a number of designers.[2] This cooperation includes sharing of designs that are half-baked, that is called uncommitted information. These features of a design activity distinguish it from the conventional notion of a transaction. Other differences include the duration of a design activity as compared to a transaction, and the notion of a design-action as opposed to an action in a transaction. Normally design activities are of long duration and span a number of sessions. In a design action the emphasize is more on the effect of the action on invariants representing design constraints. This is in

[1] The use of the word *design action* rather than *action* is deliberate.

[2] It should be noted that this is usually a controlled sharing rather than a global sharing of information.

contrast to the conventional notion of an action where the effect is seen only as operating on data items.

Irrespective of the methodology used (top-down or bottom-up), a design consists of smaller specialized designs. The specialized designs in turn consist of more specialized designs, and so on. Thus, a design activity can be modeled as a hierarchy of designs. Each design can be treated as independent, so on completion its results can be shared with other designers. A design may undergo revisions to match the evolving needs of the design team. At each stage, completed designs are retained as versions.

Designers cooperate among themselves by sharing their designs and the associated database objects that are created in the process of design. For instance, a higher level activity may create an object containing the specification to be satisfied by the lower level designs. The lower level and the higher level design activities share this specification object.

Although the progress of a design may appear similar to the concurrent execution of transactions, there are some essential differences in the characteristics and requirements of designs. These differences make the use of conventional concurrency control techniques inappropriate for design systems. Thus, alternative concurrency control schemes are required for design activities. So as not to conjure images of conventional transactions we use the word *designactivity* to distinguish a design activity from a conventional transaction.

The characteristics and requirements of a *designactivity* can be identified by analyzing the computations supported by design systems. According to Cohen [Cohe86], editing, analysis, synthesis, transformation, simulation and execution are the different types of manipulations supported by a design system. From our point of view, classifying the manipulations into the following two types is

sufficient [Goya88a].

1. Interactive.
2. Non-interactive.

Editing is an **interactive** computation punctuated with **long pauses** [Walp88]. Most edit sessions are **open-ended** activities [Pu88] and can take several hours to complete. A *designactivity* contains one or more such sessions interspersed over weeks or even months [Hask82]. These sessions are accompanied by queries or updates to the database. The results and responses of the queries and updates are used in the session. Editing being an **unstructured activity** may contain frequent **undos** [Goya88a]. On the other hand, **non-interactive** computation may include simulation, analysis etc. that are performed by executing instructions in batches. In general, computations like simulation can progress with certain **tolerance** in design and performance parameters. Results of the simulation could automatically **trigger** changes in design objects.

In addition to the above mentioned characteristics the following requirements of design applications influence the concurrency control schemes. In suggesting these requirements it is assumed that a designer is associated with each *designactivity*.

1. A model that captures the characteristics of the actual design must be developed [Kim84, Bane85, Kort88a, Beer89]. For instance, the model of a *designactivity* should capture several parallelly progressing activities of a designer [Bane85].
2. The design should support application and/or domain-specific correctness criterion (**validation**) [Kort88c] rather than the traditional *serializability* criterion based on sequence of *Reads* and *Writes*.

3. Completed results of a *designactivity* should be **globally available.** This means, that the results should be available to other *designactivities* in the same and other projects. This implies that a *designactivity* can commit independently of its parent [Pu88].

4. A designer should be made aware of any updates performed by other designers on shared objects while the design is in progress - **dynamic updates** [Maie86]. This facility may lead to *multiple reads* of the same information by a *designactivity*.

5. Access to **multiple versions** of the same object must be supported [Ditt85, Hadz85, Chou86, Katz86, Skar86, Bane87, Chou88, Kim88]. Concurrency control mechanisms can make a judicious allocation of versions to improve the concurrency [Hadz88a].

6. *Designactivities* need **private copies** of the objects [Banc85] they use. Otherwise recovery is complicated by *counter operations* [Garc83, Gare87] or *cascaded aborts.*

The above mentioned characteristics and requirements of *designactivities* have led some researchers to propose new concurrency control schemes and extensions to existing schemes [Hask82, Katz83, Lori83, Kim84, Banc85, Skar89]. The majority of these extensions propose relaxation of the serializability and correctness requirements [Kim84, Banc85, Skar86]. Extensions to locking schemes incorporate a spectrum of locks [Kort83] including hierarchy of locks [Kort88b]. *Read* and *Write* locks which permit other transactions to simultaneously read and write with notification capabilities have also been proposed [Skar86]. Extensions to optimistic schemes require the prevention of long transactions from being continually aborted as shorter transactions commit [Goya88a]. Parallelism within a transaction has been modeled by the use of nested transactions [Moss87,

Feke87]. The use of semantic contents of the operations to classify the actions and consequently to increase concurrency has also been proposed. Most of these schemes fall under pessimistic approach. These schemes do not permit access between unrelated *designactivities*, i.e. those that are not related by the parent-child or sibling relationships. *Designactivities* are not allowed to proceed in the presence of possible conflicts. The various concurrency control schemes are surveyed in Chapter 2.

In this thesis we model a *designactivity* recursively. A *designactivity* at any level can be committed and be made visible to its siblings and parents. Additional sharing is supported by the explicit identification of additional **dependents**; here after unless otherwise stated reference to dependents will include parents and siblings. Whenever a shared design or object is updated, the dependents are **notified.** Later we shall prescribe conditions under which such **notifications** are sent. In Chapter 3 we describe our notions of a *designactivity* and the object model formally.

We propose the use of an optimistic scheme and show that it suits object-oriented design applications. The use of private copies facilitates the sharing of objects by simultaneous activities. **Dynamic validation** at commit time is used to verify whether a design satisfies the validation criteria (the constraints, invariants and rules specified for the activity), on the most recent copy of the object base. The proposed concurrency control scheme allows a large set of schedules, some of which are not permitted by similar optimistic scheme [Herl87]. During the validation phase, actions of a *designactivity* are re-executed in a short time as opposed to the execution phase which is subject to interruption and pauses. Object model permits the re-execution to be performed in parallel, thereby reducing the validation time. Re-execution is made possible by keeping log of the operations, their ordering, and results. We define the compatibility between the

execution and re-execution results. This compatibility is used in defining the correctness of concurrent execution of *designactivities*. When the validation is unsuccessful, corrective actions, selective undo or controlled rollback are performed instead of aborting the complete *designactivity*. The details of our concurrency control scheme is presented in Chapter 4.

In concluding we enumerate the benefits of our approach and provide pointers for further investigation.

Chapter 2

A Survey of Related Research

This Chapter presents a survey of the research that is of common interest to the following three areas: concurrency control, object model and *designactivity* modeling. The main objective here is to present the developments that address the requirements and characteristics of a *designactivity*.

2.1. The Problem of Concurrency Control

Traditionally a transaction is defined as a *sequence* of database operations (*Read, Write*). There is only one *Read* and one *Write* per data object. A transaction cannot read a data object after it is written. Two operations from different transactions on the same data object are *conflicting* if at least one of them is a *Write*. A transaction is also treated as a unit of *recovery* and *concurrency*. This means that either the entire transaction or no part of it is executed, and transactions do not interfere with each other to produce inconsistent results.

The concurrent execution of a set of transactions, **T**, is represented by *schedules*. A *schedule* is an arbitrary sequence of operations from the set of transactions. A schedule H is *serial* if, for every two transactions t_i and t_j that appear in H, either all operations of t_i appear before all operations of t_j or vice versa. A schedule H of **T** is *conflict serializable* if there is a serial schedule H' of **T** such that every pair of conflicting operations have the same ordering in both the schedules. Unless otherwise stated the term "serializable schedule" refers to conflict serializable schedule in this Chapter. A schedule is *recoverable* if for every transaction t_i that commits, t_i's commit follows the commit of every transaction

from which t_i read. A schedule is *cascade free* if, no transaction reads or overwrites a data written by an uncommitted transaction. The task of the concurrency control system is to monitor and control the schedule so the overall correctness of the database is maintained. Conventional concurrency control schemes guarantee correctness using *serializability* and *recoverability*. A more formal description of these concepts is available in [Papa86, Bern87].

In addition to conflict serializability there are other correctness criteria whose special cases are of interest. *View state serializability* and *final state serializability* are two such criteria. Informally stated, a schedule H is view equivalent (view serializable) to a serial schedule H' , if a transaction t_i reads the data item x from t_j in H then t_i reads x from the same transaction t_j in H' , and the final writes of every data item are the same in H and H' [Bern87]. Correctness criterion defined by the view serializability is less strict than the one defined by conflict serializability. Two schedules are final state equivalent if the final state produced by the two schedules are the same irrespective of the intermediate values read and written. In general finding whether a schedule is view serializable or final state serializable is an NP-complete problem [Papa86].

2.2. Survey of Concurrency Control

A detailed study of concurrency control and recovery can be found in the references [Bern81, Kohl81, Papa86, Bern87]. The research work of relevance to this thesis can be classified into the following two areas:

1. **Concurrency Control Using Semantics:** This area of research investigates how the concurrent execution of transactions can be improved by using data type and transaction semantics. Most of these schemes do not address the issues of design application transaction.

2. **Concurrency Control for Design Applications:** This area of research investigates concurrency control schemes for *designactivity*.

The objective of the former is to use data type semantics and transaction semantics to improve concurrency. The objective of the latter is to model design applications and to develop appropriate control schemes [Bane85]. In this Chapter we briefly review the pertinent literature from the two areas.

2.2.1. Concurrency Control Using Semantics

Depending upon the semantics used, concurrency control schemes under this area can be classified into two groups:

1. Schemes using relationships that exist between different database operations defined on a specific data type.
2. Schemes using relationships that exist between different database operations issued by a transaction.

2.2.1.1. Concurrency Control Using Data Type Semantics

Conventional concurrency control schemes treat all database entities uniformly. *Read* and *Write* are the only operations defined on every object. In contrast, object-oriented databases consist of more than one data type. Each data type is characterized by a set of operations unique to that data type. This feature of object model has been used by many researchers to develop concurrency control schemes using data type semantics (semantics of the operations permitted on the data). Thus, schemes described in this section implicitly assume an object-oriented data model.

The initial research in this area was restricted to *non-compound* objects, i.e. each object contains only a set of primitive objects [Bees87]. Schwarz *et Al* [Schw84] present the synchronization issues that arise when transaction facilities

are extended to shared non-compound object types. Operation semantics, including the arguments, are used in deciding if two operations of a data type depend on each other. This *dependency information* is used to schedule operations from different transactions so that only *recoverable* and *serializable* schedules are permitted.

Herlihy [Herl88] and Weihl [Weih89] propose a decentralized concurrency control scheme. When an object receives the request for an operation from a transaction, the operation is executed on the *intention list* (*private copy*) of that object for that transaction. Before the response is sent to the transaction it is checked if the transaction can be serialized in all possible orders with other active transactions. If so, the response is sent; otherwise the transaction is made to wait. Response of the operation is also made use of in arriving at the decision. Updates to the intention list are posted on the database only when the transaction commits.

The last two schemes [Schw84, Herl88, Weih89] are pessimistic in nature because they schedule an operation only if the commit of the operation is going to produce correct result independent of the commit or abort of the other transactions. While Schwarz's [Schw84] scheme makes of use only the operation and its argument in deciding if it can be scheduled, Herlihy [Herl88] and Weihl's [Weih89] scheme makes use of the operation as well as its response.

Herlihy [Herl87] presents a set of optimistic concurrency control schemes for non-compound objects using *serial dependency relation.* Informally stated, an operation from a transaction t_i is serially dependent on an operation from another transaction t_j if the result of the former is dependent on the latter. When an object receives an operation from a transaction t_i it is executed on the intention list of the object for that transaction. When the transaction t_i is ready to

commit, the objects touched by the transaction check if the following condition is true:

$$\exists\ t_j\ ((start\_time(t_i) < commit\_time(t_j)) \wedge serially\_dependent(t_i, t_j))$$

namely whether the transaction t_i has performed an operation serially dependent on a transaction that committed after t_i started. If the transaction has performed no serially dependent operation on any of the objects, then the transaction is committed else aborted. This condition is less strict than the scheduling condition used in [Herl88, Weih89]. All the three schemes [Schw84, Herl87, Herl88, Weih89] produce a larger set of recoverable and serializable schedules than that produced by schemes that simply characterize the operations as *Read* and *Write*.

Some recent papers consider the issues related to concurrency control of *compound objects*. Badrinath *et Al.* [Badr87, Badr88] present a scheme for synchronizing operations on objects. A *directed acyclic graph* which represents objects that are accessed by an action is used to achieve intra-object synchronization. Inter-object synchronization is achieved by locking. Though this scheme handles compound objects, it is not clearly stated how insertion new objects into database will affect the scheme. Hadzilacos *et Al* [Hadz88b] present a synchronizing technique for transactions in object bases. A correctness criterion based on view serializability is presented and its implementation using locking is discussed. Unlike some of the previous schemes [Schw84, Herl87, Weih88, Badr87, Badr88] the one suggested by Hadzilacos *et Al* [Hadz88b] treats the transaction as a set of *partially ordered operations*.

The main objective of the schemes discussed in this section is to use data-type semantics. In this context, theory of operation-specific locking presented by Korth [Kort83] is important. One lock mode is associated with each database

operation. This approach is applicable to a variety of schemes presented for object-oriented databases. It allows added concurrency without increasing the overhead in the transaction manager.

The serializability conditions used by the schemes of this section are less strict than the ones imposed in the *Read-Write* model. Thus, more concurrency is possible with these schemes. However, they do not address all the issues related to *designactivities*. For instance, transactions cannot make partial commits. With the exception of [Hadz88b], other schemes [Schw84, Herl87, Badr87, Badr88, Herl88, Wieh89] treat the transaction as a *sequence* of actions. Sequence of actions do not model the *designactivity* adequately.

2.2.1.2. Concurrency Control Using Transaction Semantics.

The schemes described in the last section assume transaction as an atomic unit for failure and concurrency considerations. In design and other environments that contain long-lived transactions, such a condition is too restrictive. The following schemes relax that condition by just treating the transaction as a unit of failure atomicity. The resulting schedule may not be serializable in the traditional sense but it produces a consistent database. These schemes also make use of semantics of operations; however, the operations considered are generic database operations.

Garcia-Molina [Gare83] introduces a *semantically consistent* schedule in which transactions are classified into several types. Only transactions of certain types can be interleaved arbitrarily among themselves without affecting the consistency requirements. The set of transaction types that can be interleaved is called *compatibility set*. This means that operations of the transactions in a compatibility set can be interleaved in any order. Transaction semantics and

data-type semantics are used in the classification and in deciding the compatibility set. Recovery from failure is provided by *counter steps*. Counter steps return the database to a state that is semantically consistent with the initial state seen by the failed transaction. Garcia-Molina *et Al* [Gare87] extend the idea of multiple compatibility sets to a single compatibility set that allows transactions of all types to be interleaved.

Lynch [Lync84] introduces *multi-level atomicity*. Transaction semantics and operation semantics are captured using *break-points* and *interleaving specification*. Break-points specify a smaller unit of concurrency atomicity. Depending upon the specification, a transaction may contain as many break-points as the number of operations issued by it or may contain no break points at all. A transaction can have more than one set of hierarchically related break-point specifications. Interleaving specification describes interleavings of a set of transactions that are allowed for different break-points of these transactions. This scheme is more general than the last scheme [Gare87]. However, recovery issues are not discussed by Lynch [Lync84].

In interactive environments similar to design applications it is not possible to specify break-points, interleaving specification, counter-steps and compatibility sets. This is because *designactivities* are open-ended and progress in an ad hoc manner [see Section 3.2]. In the transaction approach, adding new transaction types requires the knowledge of the existing information (interleaving specification etc.). This is not desirable in design environments considering the diversity of knowledge that could be present.

2.2.2. Concurrency Control for Design Applications:

Predominant work done in this area is related to modeling a *designactivity*

by conventional transactions and improving the sharing among *designactivities*. Some of the characteristics and the requirements of *designactivity* listed in Chapter 1 are considered in suggesting these schemes.

A method permitting record updates by long-lived transactions without forbidding other users to access the modified records is presented by O'Neil [ONei86]. The idea behind this approach is that certain sequence of operations is executed twice to improve concurrency. Once to return the response to the transaction and once during the commit phase. This approach is an improvement of the techniques presented by Reuter [Reut82] and Gawlick [Gawl85]. The technique assumes record based *Read-Write* model. The long-lived transactions that are considered here are relatively short, batch transactions.

Some of the earlier schemes [Hask82, Katz83, Lori83] define a *designactivity* as a set of transactions involving the public database and multiple private databases. Data objects are checked out of the public system into user's private database. When a transaction is complete the updates are checked into the database. When a transaction fails, the databases (public and private) are returned to the state before the current transaction started, and not to the state before the *designactivity*. Obviously such a scheme does not model the relationship between different designs of a project. Since the complete set of data objects used by a transaction has to be checked-out. It is hard for the designers to prespecify all the data objects they would need. If a data object required by a transaction has already been checked-out then the execution of the transaction has to be delayed or aborted.

Most of the *designactivity* models suggested recently are based on nested transactions [Gray78, Moss87]. Nested transaction is a recursive concept. A nested transaction contains other nested transactions and/or traditional transactions

with sequence of actions. It is represented by a tree structure. A node in the tree represents a nested transaction. An ancestor transaction can commit independently of its descendents. Only when a transaction commits, its results are passed on to its parent. A descendent transaction can access the results from its ancestors. Timestamping and locking schemes [Moss87] are presented for synchronizing transactions within a tree and across trees. Fekete *et Al* [Feke87] prove the correctness of the locking protocol presented in [Moss87].

Kim *et Al* [Kim84] have suggested an augmented scheme that refines the notion of check-out environment [Hask82, Katz83, Lori83]. This is achieved by introducing the notion of semi-public databases. A *designactivity* has semi-public database into which it may place the design objects it has updated. Once a *designactivity* places an object in its semi-public database, other authorized *designactivities* may check it out. A *designactivity* that checks out of another *designactivity's* semi-public database becomes a child of that *designactivity*. A *designactivity* may access both private and public databases. When a child *designactivity* of a *designactivity* commits, the changes to the semi-private database of the parent and the public database are synchronously committed. The public system manages both the public database and semi-public databases of all *designactivities*. This scheme does not guarantee recoverable schedules. For instance, if a parent *designactivity* decides to abort, then some of the changes made by its child are also lost. This means that the public database contains only part of the changes committed by the child. Synchronization techniques suggested in Kim *et Al* [Kim84] use locking, and the *Read-Write* model. In this scheme, as in [Hask82, Katz83, Lori83], the user must prespecify the objects that will be used. The other limitation of the Kim *et Al* [Kim84] scheme is its inability to model the parallelism present in a typical *designactivity*.

Bancilhon *et Al* [Banc85] and Korth *et Al* [Kort88a] eliminate this limitation. They model the *designactivity* using a conceptual hierarchy of multiple types of transactions instead of a sequence of conventional transactions.

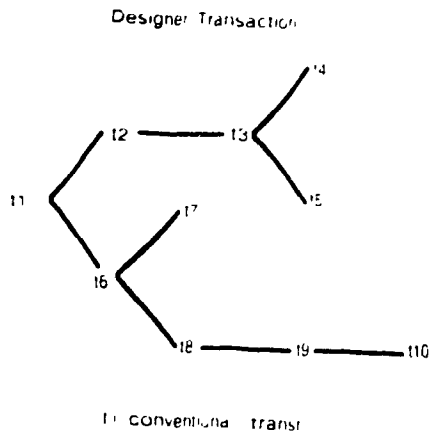
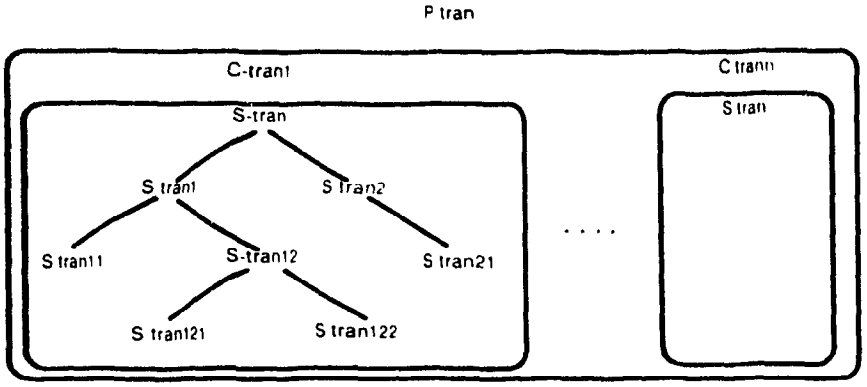


Figure 2.1. An Example of Designer's Transaction

The top level of the hierarchy represents the *designactivity*. Each *designactivity* may consist of a set of *cooperating transactions*. Each cooperating transaction is a hierarchy of *client/subcontractor* transactions. A client/subcontractor exists solely to work on behalf of another client/subcontractor transaction. Each client/subcontractor transaction consists of a set of *designer's transaction* which in turn consists of a set of conventional transactions (see Figure 2.1). Each conventional transaction is initiated from a window of the designer's workstation. Figure 2.2 illustrates an intuitive model of this scheme. The idea of public and semi-public databases is still used in implementing the sharing among design activities. Though the model of the *designactivity* is more powerful than previous schemes [Hask82, Kat83, Lori83, Kim84], there is no marked deviation from the *Read-Write* model and the locking scheme. The correctness criterion used in this



P-tran - Project Transaction
C-tran - Cooperating Transaction
S-tran - Client/Subcontractor Transaction

Figure 2.2. Intuitive Model of CAD Transaction[Banc85].

scheme is less strict than the previous schemes [Hask82, Katz83, Lori83, Kim84]. There is no sharing between two transactions that are not components of the same group of cooperative transactions. This limitation is inherent in the *designactivity* model because no two transactions above the level of client/subcontractor transactions can share incomplete objects. This approach captures the transaction semantics or *designactivity* semantics by specifying a set of conditions that are to be preserved by the transaction. These conditions are specified at the transaction initiation time. They are independent of the other transactions and the specifi of the conditions does not need any additional knowledge.

Skarra *et Al* [Skar86, Fern89, Skar89] also describe a concurrency control model of cooperating transactions. In this scheme a *designactivity* is represented recursively using nested transactions and multi-level commit. Each node in the

hierarchy can be an atomic transaction similar to the nodes of nested transactions or it can be a *transaction group* (*TG*). A TG can recursively contain other TGs. The *root* of the hierarchy represents an atomic transaction. Each transaction group G produces an operation sequence, S, comprised of the interleaved sequence of its members. S is correct under the semantic criteria that G defines. The root of every TG subtree has an atomic transaction parent. Operations from atomic transactions are scheduled only if there is no dependency between operations. The dependency relation used is an extended form of that used by Herlihy [Herl87] and Weihl [Weih89]. Here an operation a that is delayed due to its dependency relation with an already scheduled operation, b, may be scheduled even before b is committed. This is possible when the sequence of already scheduled operations is commutative with the operation to be scheduled. However, it is not clear how this scheduling can be efficiently extended to more than two transactions. There is no provision for sharing completed objects across different subtrees in the hierarchy.

Most concurrency control schemes use the *Read-Write* model. More and more CAD systems are designed using object-oriented databases. Using the *Read-Write* model with such systems will not allow us to exploit the benefits of object-oriented systems. Despite their sophisticated nested structure [Banc85, Skar89], aborting a transaction at any level implies aborting all the enclosed activities. In addition, interaction with other concurrent activities is limited to the sibling transactions that are enclosed in the same parent transactions [Banc85, Skar86, Fern89, Skar89]. All the schemes discussed in this section make use of locking based pessimistic schemes. Limitation of pessimistic schemes in design environment is discussed in Chapter 1. Recently some researchers have suggested alternatives to, or relaxation of, serializability as the correctness criterion [Kort88c, Skar89].

Chapter 3

Model

The first section of this Chapter presents our model of the *designactivity*. Design being an opportunistic activity, is difficult to model. It is even more difficult to define models for concurrency control because a typical design includes access to other incomplete or complete designs and access to the object base. A model of the *designactivity* must include all those factors that are common to the *designactivities* of different application domains. This model can then be used to develop a control scheme for the sharing and completion of design activities. Control schemes developed for *designactivities* must be general than the traditional concurrency control schemes.

The second section of this Chapter presents our object model, and the architecture of the distributed system. We discuss how the entities of our *designactivity* model are mapped to our object model.

3.1. Design Activity Model

Normally *designactivities* start by identifying some initial pool of design candidates. Some of these are identified for reuse, modification etc. Design being an iterative activity proceeds in steps of refinement. However, the levels of detail introduced in these steps are not uniform. At any stage of the *designactivity* - even when a design has been completed - the designer may make modifications, thereby creating a new design new versions of the design. A very trivial example is that of a generic stack which can then be modified for individual types and implementation languages and systems.

A *designactivity* is domain-specific. Even within an application two different

designs can have certain distinct validation conditions. Thus, no global criterion for the validation of designs can be specified. It is, however, possible to model these validation and correctness criteria as a set of constraints on the *designactivity*. Similarly rules for sharing, borrowing and copying can be tailored to different domains or problems. One of the important features of a *designactivity* is the ability to save partly completed designs for future use. Organizations have different policies on their use, availability etc. Again these can be modeled by rules or constraints.

A *designactivity* may be modeled as a number of pseudo independent subactivities; the subactivities are themselves so modeled. Designers are assigned these *designactivities* or subactivities. In general, because *designactivities* depending on each other may need to be separated, they are said to be pseudo independent. In fact the spectrum of dependencies among the various subtasks of a given *designactivity* would range from complete independence to complete dependence. In such a design environment, controlled sharing of information, for example partially completed designs may be the only way to control design progress.

The above division of a *designactivity* can be defined recursively:

Definition 3.1: A *designactivity* contains other *designactivities* and/or **transactions.**

Definition 3.2: A **transaction** contains a set of **action invocations.**

Informally stated, an action invocation is an operation performed on a data object. A *designactivity* can be represented by a tree diagram, where each non-leaf node represents a *designactivity* and each leaf node represents a transaction. The root of the tree is referred to as the *project activity*. It represents the way a project is divided into various *designactivities*, as well as the *dependencies* among

different activities in the traditional top-down and bottom-up designs. The terminologies *child, parent, sibling, ancestor,* and *descendent* are used consistent with the tree structure. Two *designactivities* are said to be *unrelated* if they are different and neither is a descendent of the other.

Each *designactivity* is characterized by two types of constraints, namely **validation constraints**, and **visibility constraints**. Validation constraints specify the conditions under which a *designactivity* is deemed *completed* (committed). Design requirements, design rules and correctness criteria of the conventional concurrency control scheme are some of the examples that come under validation constraints. Visibility constraints specify the conditions under which a *designactivity* allows itself (or its results) to be used by other *designactivities* and/or conditions under which it uses the other *designactivities* (their results). One such constraint used in many design activity models [Bane85, Skar89] is: when a design and all its children are completed, it can be used by its parent and siblings. Such a usage is restricted to the parent and siblings of the design.

The solid arcs of the *designactivity* tree form the natural association among different activities and represent the visibility constraints among them. Dotted arcs between unrelated activities represent *dependency* between two unrelated activities (see Figure 3.1).

The completion of a *designactivity* is controlled by the validation constraints. If a *designactivity* cannot be completed due to the violation of certain constraints then the designer becomes aware of the violation. The designer can make the necessary changes and attempt to commit iteratively until the *designactivity* conforms to the validation constraints. When a *designactivity* is completed it is made visible to the other activities specified by the visibility constraints. These

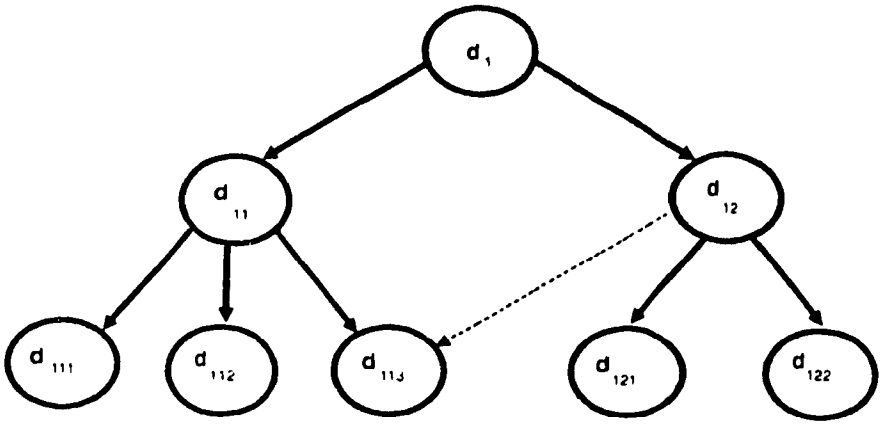


Figure 3.1. Tree Representation of a Design Activity.

constraints may not only specify who gets to use the *designactivity* but also the condition of usage. For example, visibility constraints can be used to model notification [Fern88]. Creation of versions, dynamic updates, and storing invalid design are possible using the validation and visibility constraints.

The runtime checking of these constraints can be potentially very expensive. There are however a set of restricted constraints that can be efficiently satisfied - *weakly positive*[3] [Scha78], and *completely bound*. These are powerful enough to model a very large set of practically occurring constraints. The use of *alerters* and *triggers* [Eswa76, Bune79], which are standard features of an increasing number of new database systems (e.g., POSTGRES [Ston86a], Eden [Blac85]) can also be used to efficiently implement consistent enforcement mechanisms.

This model of the *designactivity* is simple and general. Its power can be demonstrated by deriving other *designactivity* models by adding appropriate

[3] A formula R is weakly positive if $R(x,\ldots)$ is logically equivalent to some conjunctive normal form formula having at most one unnegated variable in each conjunct

constraints to it. This is illustrated in the next Chapter. The generic design model introduced here can also be used to specify other non-design activities like *groupware system* [Elli89] that need to share information under different conditions.

As suggested earlier, validation constraints include different serializability criteria, design requirements, design rules etc. This means some of the validation constraints like correctness conditions can be checked as the *designactivity* progresses using the existing concurrency control techniques. Some of the other conditions like design rules, performance requirements can be checked at the end of the design when the designer thinks the *designactivity* is ready to be completed (committed). In the next Chapter a validation scheme that does the validation at the end for all types of constraints is presented. It is also shown, using an implementation of the scheme how the validation and visibility constraints are used. This scheme satisfies the requirements of design applications and traditional interactive applications.

3.2. Object Model

The object model described here does not make any assumptions about the method-sharing amongst different objects. That is, our object model is independent of **delegation** [Libe86] or **inheritance** [Stei87, Unga87]. Due to this reason the concurrency control schemes described here can be used with any object model that assume delegation and/or inheritance. The following two approaches to object-oriented computing are reported in the literature [Yoko87]. This thesis follows the second approach.

1. In addition to *objects*, the notion of *processes* or *monitors* is introduced in Yokote *et Al* [Yoko85]. This is also the approach taken in

Smalltalk—80 [Gold83].

2. The two entities, *objects* and *processes* can be combined into one single entity by conceptually associating a process with each object and making the object and the corresponding process indistinguishable from each other. This is the approach taken by *Act*—1 [Libe81], Goyal *et Al* [Goya87], and *ABCL*/1 [Shib87, Yone87]. This allows every entity in the system to be treated uniformly.

Our object model is based on the one presented in Goyal *et Al* [Goya87] and Hadzilacos *et Al* [Hadz88b]. The idea of object management and communication between objects is derived from Goyal *et Al* [Goya87]. Object structure and operation characteristics are derived from Hadzilacos *et Al* [Hadz88b].

Definition 3.3: An **object base** is a set of **objects.**

Objects are persistent, uniquely addressable and autonomous information storing and processing agents. Each object either *models* the behavior of an abstract entity (*designactivity*, design specification, transaction etc.) or *represents* a physical entity (processor, printer etc.) itself [Booc86]. Objects are classified into several **classes.** Objects of the same class have a common behavior. Classes are themselves modeled as objects. A class object consists of the declarations and reentrant code for the operations of that class. It also contains the code for initialization, concurrency control, recovery etc. These code modules are used during object (process) execution.

Definition 3.4: An **object class C** is a 2-tuple **<V,A>**, where **V** is a set of **variables** and **A** is a set of **actions.** An **object instance** of the class **C** contains a **state** and an **identity**, where the **state** is a mapping of the variables of **C** to values, and the **identity** is a value that uniquely identifies the object.

The values of the variables are drawn from the domains of object identity, and primitive objects (Strings, Integers etc.). The state of an object may change over the course of time. However, the value of its identity remains the same from the time the object is created until it is deleted. Conceptually, the state of an object contains references to other objects rather than objects themselves, although primitive objects may be freely replicated to replace references to them in an implementation. Values that are assigned to state variables of an object are also called *attributes.* The *state of the object base* is a mapping from the set of object identities in the object base to their respective states.

Informally stated, an action contains a sequence of operations (steps) on the state variables of the object. An action is initiated through an **action invocation** or *invocation* in short and the result is returned as a **response.** On receiving a message with an invocation, the object executes a set of prespecified *steps.* A step could be a *local operation* that makes use of the state value of the object or could be an *invocation* to some other object. This means that an action can invoke actions on other objects, and these in turn can invoke actions on other objects and so on. After performing all the steps in the set, a *response* is returned to the sender of the invocation. The action invocation and their responses can be termed as *events.* Events also encompass *triggers, daemons* etc. An action execution begins when the message containing the corresponding invocation is received by an object, and is completed when the object that sent the message gets the response.

Definition 3.5: An **action** A is a sequence of **steps** and is a unit of **consistency,** and **atomicity:**

a. an action execution is initiated through an *action invocation*

b. a **step** could be either a **local step** or an **action invocation** which in turn initiates an action execution at the receiving object.

The term "unit of consistency" implies that if an action were to be executed in *isolation* on a consistent object base it will perceive and produce a consistent object base. The term "unit of atomicity" implies that either the entire action or no part of it is executed.

The action invocation K is a 5-tuple $<t,p,Q,A,k>$, where t is the identity of the original action on whose behalf this action is invoked, p is the identity of the object called the **invoker** sending the message containing the action invocation, Q specifies the set of objects that should receive the message, A is the action to be invoked on the objects in Q, and k is used to identify the invocation. Though in general Q can be a predicate (defining a set of objects), at this point we simply treat Q to be an enumeration of object identities.

The **response** R to invocation K is a 5-tuple $<t,q,p,r,k>$, where t, and p have the same meaning as in the invocation K, q is the source object of the response, and r is the value returned from q for the invocation K. r is obtained as a function of the following:

1. Results of the local steps in action A of K.
2. Responses returned to the invocations in A of K.

Definition 3.6: A **Local step** a of an object o_i is a pair (f_a, g_a) where $f_a : S \longrightarrow R$ is a function from the set of states S of the object to the set of **results** R and $g_a : S \longrightarrow S$ is a function from the set of states S to the set of states S of the object.

Local steps are atomic operations on the variables of the objects [Beec87]. Thus, $f_a(s_i)$ specifies the result of the local step a when executed on state s_i and

$g_a(s_i)$ specifies the new object state caused by the local step a when executed on state s_i.

A transaction is an action of a special class of object called *transaction-manager*. A unique transaction-manager object is associated with each transaction execution. Function of a transaction-manager object is to manage the execution of the associated transaction. In addition to transactions there are other actions (abort, roll-back etc.) associated with the transaction-manager class. A transaction-manager object can also invoke certain distinguished actions, including *start-commit* and *commit* in other objects.

Due to its recursive nature, an action execution can be represented by a tree, referred to as the **action execution tree.** The action execution tree of a transaction is called **transaction execution tree.** In this tree the directed edges represent actions and the nodes represent objects. Edges are from invoker to receiver nodes. Recursively each sub-tree represent an action execution tree. The arc between the parent and child represents the bidirectional communication between the two. Figure 3.2 illustrates an example of the tree representation described above. In the transaction execution tree, the root (level 0) is associated with the transaction object. The tree has at least two levels.

Definition 3.7: An action B of object o_i is executed *on behalf of* an action A if one of the following conditions hold:

1. execution of A invokes B in o_i.
2. execution of C invokes B in o_i, where C of o_j is executed *on behalf of* A.

Definition 3.8: An object is a participating object of an action A if it executes one or more actions *on behalf* of A.

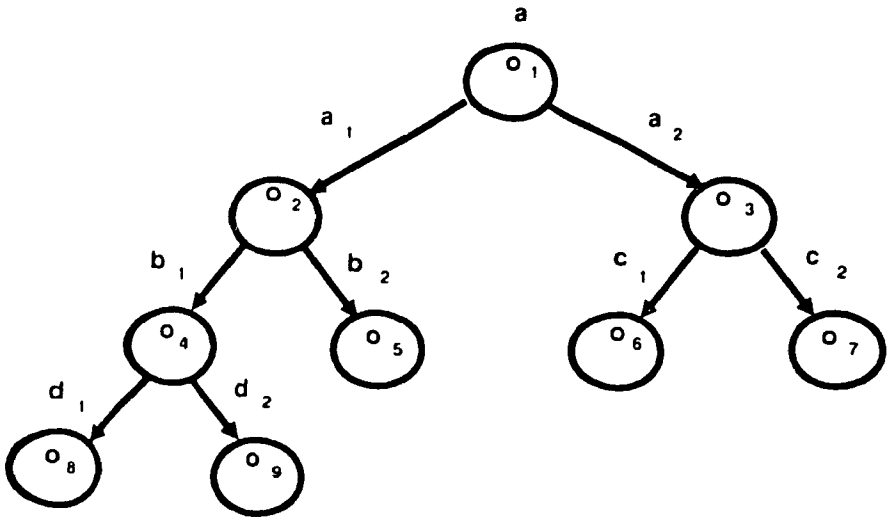


Figure 3.2. Tree Representation of an Action Execution.

Definition 3.9: A step a_i is executed *on behalf* of an action A if one of the following conditions hold:

1. a_i is a step in A
2. a_i is a step in B, where B is an action executed *on behalf* of A.

By imposing a total order on the set of steps of an action A one can get the traditional transaction of sequential steps. In traditional concurrency control systems the following definitions are used.

Definition 3.10: Given a set of transactions $\mathbf{T} = \{t_1, t_2, \ldots, t_n\}$, **a schedule** Z is a 2-tuple $<E, S_0>$ where:

1. E is a sequence of steps in $\mathbf{T}$ (E is the union of the steps executed on behalf of t_i's, where $1 \leq i \leq n$).

2. Every pair of steps executed on behalf of a transaction t_i $(1 \leq i \leq n)$ is ordered consistently in E. That is, if step a precedes (succeeds) step b in t_i then a precedes (succeedes) b in E.

3. S_0 is the initial state of the object base.

Definition 3.11: Two schedules $Z_1 = (E_1, S_1)$ and $Z_2 = (E_2, S_2)$ are **equivalent** iff

1. $S_1 = S_2$.

2. Schedules Z_1 and Z_2 have the same set of steps.

3. For every local step a, $f_a(s_{i1})$ is *equivalent to* $f_a(s_{i2})$, where a is a local step of object o_i; s_{i1} and s_{i2} are the states of o_i in Z_1 and Z_2 respectively when a is executed.

The use of the phrase "equivalent to" instead of "equal to" in condition 3 makes this definition more general than traditional definitions based on equality of results. This property is used in Chapter 4 to describe an implementation that is not restricted to equality of results.

Definition 3.12: A schedule Z is a **serial** schedule iff for any two transactions t_1, t_2 in **T** either every step executed on behalf of t_1 precedes every step executed on behalf of t_2 or vice versa.

Definition 3.13: A schedule Z is **serializable** iff it is equivalent to a serial schedule.

3.2.1. Architecture

Every object in the system has an associated object manager, OM. The functions of an OM include keeping track of action invocations responses, and

determining the next action to be executed. The latter function of the OM is delegated to the scheduler in each OM. In systems where the OM creates a new private copy it also creates a copy of the OM(s) to schedule actions on the copy.

In conventional object-oriented systems, the actions are executed on a first-come first-served basis, namely the OM(s) will simply be a FIFO queue. In others, the OM(s) can provide different mechanisms. For example, a dequeue action on an empty queue could be delayed until an appropriate time. The object manager is also responsible for sending responses to the distinguished invocations issued by the transaction manager. For instance, any of the following two responses can result from a **start-commit** invocation.

1. **Ready-to-commit:** This is one of the two responses returned for the invocation Commit-Request. It informs the transaction manager that the object manager is ready to commit the changes requested by the transaction.

2. **Abnormal-event:** This is the other response returned for the invocation Commit-Request. It informs the transaction manager that the object manager can not commit the transaction, and also instructs the transaction to send an abort message to its participating objects. In the case of traditional databases abnormal-event is called abort.

Appropriate responses are defined for the other distinguished invocations.

To map the *designactivity* model to the object model we make the following associations: each *designactivity* is associated with an object similar to the transaction-manager. This object can issue all the privileged actions that are invoked by a transaction-manager object. In addition, it has got the privilege to create other *designactivities* and objects.

Our model of the distributed system consists of some nodes that

communicate using messages over an abstract communication network, as shown in Figure 3.3 [Moss87]. Each node consists of a set of processor elements (PEs), and some memory (see Figure 3.4). Each node is managed by a node-manager object which is resident on the PE called the manager PE (MPE). Each PE is managed by its processor element manager (PEOM). PEOM is responsible for scheduling other OMs on the PE. Failure of communication to this node or the failure of the node itself is modeled by the failure of the MPE. No assumption is made about how the other PEs belonging to the node are connected among themselves and to the PE. Failure of one or more of the PEs does not lead to a node crash. Every pair of nodes can communicate with each other either directly or indirectly using the network.

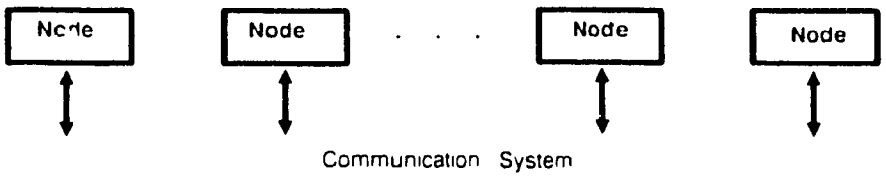


Figure 3.3. Model of the Distributed System.

Each abstract object (data object, transaction) belongs to a single node [Moss87, Dall85] known as its *home site*. The home site of an object has complete control over that object and its manipulation. This makes the management of a single object easier. However, this does not prevent transactions from sending

invocations to objects resident on other nodes. Any replication or movement of an object is explicit.

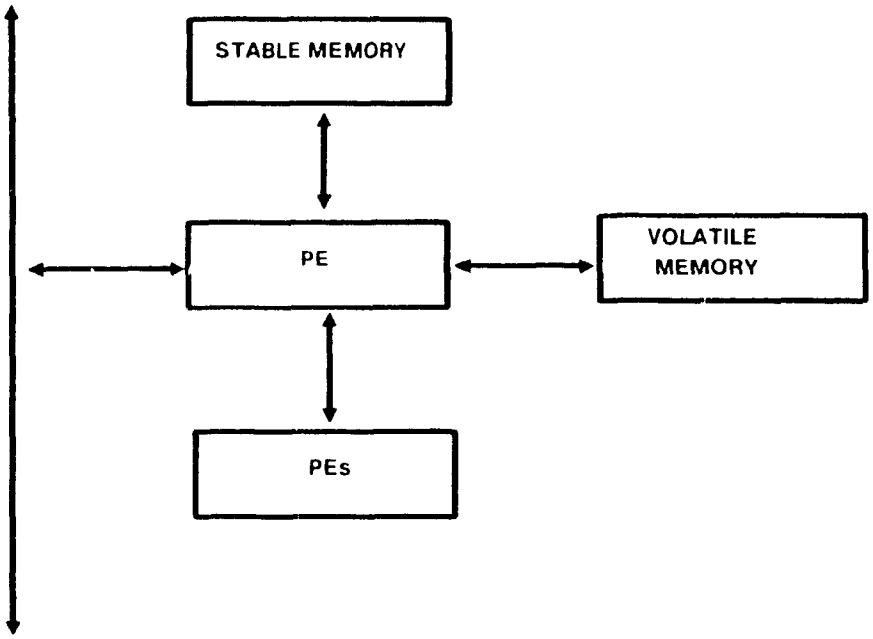


Figure 3.4. Model of a Node.

All message transmissions in our computation model are *asynchronous*. That is, an object can send a message whenever it likes, irrespective of the state of the target object. Though message passing in a system of objects may take place concurrently, we assume message arrivals at an object to be linearly ordered. Two concurrent messages arriving at the same object will be arbitrarily ordered.

A communication system with the following characteristics is assumed:

1. Every message reaches its destination.
2. Virtual circuit is assumed between a source and a destination. Hence the messages arrive in the order in which they are sent.
3. Multicast Communication is assumed.

As soon as a dormant object becomes active, the OM initializes all internal variables, structures etc. of the object. Currently active objects can be visualized to be resident on one or more of processor element (PE) objects. Chapter 5 discusses the role played by the node-manager object in the execution of an action.

3.3. Summary

Unlike the existing *designactivity* models, the model presented in this Chapter does not impose any restriction on sharing completed results among unrelated *designactivities*. Validation constraints are used to specify the conditions that are to be satisfied for a *designactivity* to be correct and complete. These constraints can be used to capture correctness criteria that may be application dependent, or application independent. Serializability is one such application independent condition. Thus, the model can capture a wide variety of correctness criteria.

In general the validation process can be complex needing exponential time to find whether all the constraints are satisfied for a *designactivity*. However, the validation can be carried out effectively for certain subclasses of validation constraints such as weakly positive [Sche78] and bounded (variables are bound to some constants).

We associate a transaction-manager object with each transaction to communicate with other objects to achieve the required correctness. This allows communication among transactions, and consequently an implementation of a distributed concurrency control scheme.

Chapter 4

Dynamic Validation

In interactive decision-support applications there is considerable time (thinking time) between successive invocations of a transaction. A number of these invocations may be discarded due to various reasons before the transaction is committed. In this type of environment, pessimistic schemes are too restrictive. Optimistic schemes like forward validation [Herl87] are not efficient because a transaction could be aborted in favor of other conflicting transactions. These transactions may or may not commit eventually. Even if they commit, there is no guarantee that they still contain the conflicting actions that caused the abort. The backward validation scheme [Herl87], where the decision to abort a transaction is made at its commit time is suitable for this type of environment, because, when a transaction wants to commit, there is no uncertainty in what it wants to commit. Even if it is aborted, it is done against the already committed transactions.

Re-execution of an entire transaction at its commit time will be faster because of the absence of thinking time and the cancellations of invocations. This idea forms the basis of the dynamic validation scheme. First a set of application independent validation constraints are derived to demonstrate the use of dynamic validation scheme for a set of transactions. An implementation of the scheme using locking is discussed and its correctness is proved. We show the power of the generic model by deriving some transaction models from it. Finally the applicability of this scheme to the generic model, and the performance of the scheme in comparison with other schemes are discussed.

The execution of a transaction is done using private copies of the object. Then we employ two-phase commit. In the validation phase the steps of the

transaction are *re-executed* on the most recent copies of its participating objects. Due to the following reasons the re-execution takes only a fraction of the execution time.

1. No message (invocation/response) is sent from the OM to other OMs on behalf of the re-execution. Each OM assumes the same responses as in the execution phase.
2. The re-execution is carried out concurrently in all participating objects.
3. The re-execution in an object is carried out only if the new state is different from the previous state of the object. In fact, as long as the result of any local reads remain unchanged, there is no need for re-execution. This observation leads to a more efficient implementation of the algorithm which keeps track of changed state variables.

In the next phase (we shall refer to it as the *termination* phase) appropriate commit, abort or roll-back actions are taken based on the outcome of the validation phase. The steps of the commit algorithm can be simply stated as follows.

Commit Start:

I. Validation Phase:

1. TM sends a commit-request to all the participating OMs.
2. Each participating OM re-executes all the local steps that were executed on behalf of that transaction on a new private copy corresponding to the current state of the object. This copy is called the *re-execution copy* of the transaction with respect to that object.
3. During this re-execution, every OM checks if the results (or effects) obtained during the original execution and the re-execution are *compatible* (validation

condition) for every local step. Any constraints specified for the steps or the objects must be satisfied.

4. If the results are compatible then the OM sends a ready-to-commit message directly to the TM else sends an abnormal-event message.

II. Termination Phase:

1. If the TM receives the ready-to-commit messages from all participating OMs, it sends the commit message to all these OMs. The current state of each of the participating objects is changed to the modified re-execution copy.
2. If the TM receives any abnormal-event message then the appropriately specified action is taken. In conventional schemes this is normally abort. If the transaction aborts, then both the execution and re-execution copies of the participating objects are discarded. In *designactivities*, the designer may choose to abort, roll-back to some appropriate point (namely, *undo* the effects of some of the steps), or perform additional actions, including the **save-invalid** action.

Unless otherwise stated a commit is treated as a global commit (implicit visibility constraint). Since execution and commit are performed using private copies, recovery from an aborting transaction is trivial. When a transaction commits, its re-execution copies (including the effects of the re-executed steps) become the current copies of the participating objects. The use of private copies prevent cascading aborts.

4.1. Dynamic Validation Scheme

The methods presented in this thesis are based on the following correctness principle: In an object base, concurrent execution of the transactions is serializable

if their order of execution is consistent in each participating object. That is, every pair of transactions t_i and t_j commit in the same order in every common participating object. The serialization order achieved here is the order in which the transactions commit.

In most of the existing concurrency control schemes, two local steps a, b of an object are deemed conflicting if one of the following conditions is false for *at least one state* s_t of the object o_t.

$$g_a(g_b(s_t)) = g_b(g_a(s_t)) \tag{4.1}$$

$$f_a(g_b(s_t)) = f_a(s_t) \tag{4.2}$$

$$f_b(g_a(s_t)) = f_b(s_t) \tag{4.3}$$

where $f_a : S \rightarrow R$ and $g_a : S \rightarrow S$ represent functions representing the results and new states, respectively, as defined in Section 3.1. Transactions with conflicting steps are not allowed to be concurrently executed at other states even if all the above three conditions were true. This definition of conflict was not only adequate but was also needed to construct efficient schedulers for the existing record based models in commercial applications.

With the object model and in particular design applications, these conditions can be relaxed and still efficient schedulers can be implemented. The only requirement is that

$$f_a(s_t) = f_a(s'_t) \tag{4.4}$$

where the original execution of the operation is on a copy of the object's state s_t while the commit is performed on a copy of the object's current (at commit time) state s'_t. Thus, the equation 4.4 checks if the two states of the object o_t are equivalent enough to return the same result for a. Since s'_t is the result of already committed transactions there is no need to check condition 4.1. Only

condition 4.2 (or 4.3) must be checked for every local step of a transaction to guarantee serializability. Obviously condition 4.4 is less restrictive than the conditions given by 4.1 to 4.3.

This optimistic dynamic validation scheme is similar to the backward validation schemes of [Herl87, Kung81]. In backward validation schemes [Herl87, Kung81] conflict is identified using static information that is insensitive to the state of the object. In contrast, in our scheme invalidity of actions is identified during the commit phase. Since the validation is performed with respect to the current state of the object we refer to our scheme as the *Dynamic backward validation* . It should also be noted that a transaction acquires a copy of an object dynamically when an invocation is sent to the object for the first time. Serializability can be guaranteed by providing arbitration schemes during this commit phase, for example, by locking as described in the following section.

4.1.1. Dynamic Backward Validation using Locking (DBVL)

In this section we shall use locking to provide the arbitration mechanism for concurrent commits. Normally an object manager can concurrently process the commit-requests of more than one transaction, provided, all these transactions are read-only with respect to its object instance. The more interesting and useful situation is where some arbitration is required. One way of achieving arbitration is through the following locking protocol.

1. A transaction is validated in an object only if it holds a validation lock on the object.

2. If a transaction requests a validation lock on an object, the request can be granted only if no other transaction holds a validation lock for the same object.

3. When a transaction commits or aborts, its locks are released.

4. All transactions not in their commit phase, have access to their private copies of the objects. They are unaffected by this locking. Also new private copies can be created for any new transaction even if the object is locked.

When t_j wants to perform the validation phase of its commit, the participating object managers lock their respective objects. This means only t_j can commit and the other transactions wanting to commit should wait. This protocol could lead to deadlocks. A distributed deadlock detection and resolution algorithm, e.g. Moss [Moss87], can be used to resolve this problem. Deadlock can be resolved by asking a transaction to release all its locks and restart its commit phase (instead of restarting the entire execution of the transaction). A timestamp-based deadlock avoidance protocol can also be used [Goya88b].

To improve concurrency during the commit phase, two types of validation locks can be supported : *Read* and *Write*. *Read* locks are compatible with each other, while *Write* locks conflict with other *Write* and *Read* locks. Each object manager assigns the appropriate type of locks for a committing transaction.

4.1.1.1. Correctness of DBVL.

The following definitions of compatible responses and results are used to prove the correctness of DBVL. For each database object the object designer specifies the minimal response (or result) compatibilities criterion for each action. These criteria can be strengthened or weakened by individual transactions. We can model this by a boolean function **compatible:**

$$Compatible: Action \times Response \times Response \rightarrow \{True, False\}$$

Application dependent or independent criteria can be used to define the *Compatible* functions. For instance, equation 4.4 treats two results to be compatible if they are equal. This definition of compatible results is independent of application and even data type. The following type independent compatible function defines a relation that contains the relation defined by equation 4.4:

1. Two *results* $f_a(s_i)$ and $f_a(s'_i)$ of a local step a of an action A are compatible if the responses and invocations that appear after the execution of a in A remain equal whenever $f_a(s_i)$ is replaced by $f_a(s'_i)$ and vice versa. Let 4.4' denote such a compatible function.

$$Compatible'\ (a, f_a(s_i), f_a(s'_i)) \qquad (4.4')$$

2. Two *responses* of an invocation K are compatible if they are equal.

A correctness proof for the DBVL scheme under these criteria is presented below.

Lemma 4.1: Two responses received by an invocation $<t,o_j,o_i,A,k>$ in two different states S_1 and S_2 of the object base are compatible, if the following conditions are true:

1. Every local step a of o_i gets compatible results in S_1 and S_2.
2. Every invocation K from o_i to other objects (on behalf of a) gets compatible responses in S_1 and S_2.

Proof: The response to an invocation is a function of

i. results of the local steps in A and

ii. responses to the invocations in A.

It follows from the definition of compatible results and responses (4.4') that the

responses of A are equal in S_1 and S_2. Two responses are compatible if they are equal. □

Theorem 4.1: If the same transaction is executed in two different states S_1 and S_2 of the object base, the transaction will get compatible responses for every one of its invocations in both the states, if every local step executed on behalf of the transaction gets compatible results in these two states.

Proof: The theorem is proved by induction on the height of the transaction execution tree.

When there are only two levels (level 0 and level 1): The transaction sends an invocation to an object, the object executes only local steps on behalf of the invocation and returns a response to the invoker. If every one of these local steps gets compatible results in the two different states of the object then compatible responses will be returned in these two states.

When there are n levels: By the inductive hypothesis all invocations sent from objects in level one get compatible responses (in S_1 and S_2). This means all the invocations which are sent from an object o_i in level one on behalf of the invocation a (from level 0) will receive compatible responses. Every local step of actions executed on behalf of a also gets compatible results in S_1 and S_2. Hence, a will receive compatible responses in these two states by Lemma 4.1. □

Now, let us consider the execution phase and the re-execution phase (commit phase) of a transaction. A successful commit requires that the result of every local step be compatible with the one from the execution phase. Hence, the re-execution is a high speed replay of the execution as far as the invocations and responses are concerned - in fact the re-execution does not send any messages, and assumes the

same invocation and compatible responses as the execution phase. Only local steps are executed in parallel in all the objects involved. The state may be different when the commit phase starts, and will be mapped appropriately by local steps to the final state. Hence the overall execution of a (successfully) committed transaction is a combination of invocations and responses from the execution phase plus state changes from the commit phase. This overall execution is exactly the same as a (full) execution of the transaction - running alone (serially) - from the same object base state as the starting state of the commit phase. The locking protocol used during the commit phase enforces serializability.

Lemma 4.2: Let t_1 and t_2 be two transactions, and let them both invoke actions on objects o_1 and o_2 (among others). Then, using DBVL, t_1 and t_2 commit in o_1 and o_2 in the same order.

Proof: Such an ordering is enforced by the locking protocol. □

Now we can prove the main result of this Chapter. Note that our definition of serializability (see page 30) is based on the notion of equivalent (or compatible) results, and is less restrictive than the classical serializability criterion.

Theorem 4.2: DBVL enforces serializability for compatibility function represented by equation 4.1.

Proof: Let $G = (V,E)$ be a graph, V consists of nodes representing transactions, and $(t_i, t_j) \in E$, the set of edges, iff there is an object o where t_i commits before t_j. By Lemma 4.2, G is acyclic, and induces a partial order between transactions. Take a total order consistent with the partial order induced by G, then the execution enforced by DBVL is equivalent to the serial execution represented by the total order. Here two results are defined to be equivalent if they are

compatible according equation 4.4'. □

4.2. Performance of Dynamic Validation

To evaluate the performance of the dynamic validation scheme we compare the concurrency and modularity offered by several schemes. An analysis is made to justify the claim that the re-execution of the transaction takes only a fraction of its execution under the specified conditions.

4.2.1. Concurrency

The following example illustrates that there exist schedules allowed by the dynamic validation scheme that are not allowed by other established schemes.

Example 4.1:

Two transactions t_1 and t_2 concurrently perform the steps Debit(90$) and Debit (100$) respectively on the same account object whose *Balance* is equal to 200$. Debit operation returns 'Over Draft' if the balance is smaller than the debit amount else returns 'OK'. The *Balance* is also updated to reflect the current balance if the response is 'OK'. According to the conflict definition of [Hadz88b] two debit steps from different transactions are conflicting at certain state values, and thus should never be allowed concurrently.

Locking Scheme:

If transaction t_1 (t_2) were to lock the account object first then the transaction t_2 (t_1) would wait until t_1 (t_2) completes.

Time stamping:

If we assume the time stamp of t_1 (t_2) is older than that of t_2 (t_1) and if t_2 (t_1) commits before t_1, (t_2) then t_1 (t_2) will be aborted [Bern87]

because two debits are conflicting.

Optimistic - forward validation:

If t_2 (t_1) wants to commit, then the responses [Herl87] of the steps of the active transactions on the account object are compared. Since t_1 (t_2) and t_2 (t_1) have the same response (OK), t_2 (t_1) will be aborted.

Optimistic - backward validation:

Assuming t_2 (t_1) commits before t_1 (t_2,, the latter will be aborted when it attempts its commit [Herl87].

Dynamic Validation Scheme:

During the execution phase t_1 and t_2 are going to get OK as their responses. Irrespective of the order of their commits t_1 and t_2 will get the same response from the account object during the re-execution phase:

If t_1 decides to commit first then the re-execution of t_1 on the current state of the account object will make *Balance* equal to 110$. Later when t_2 wants to commit it is re-executed on the account object whose current balance is 110$ (not 200$). Even with this balance, the account object will be returning the same response ('OK') to t_2. This means condition 4.1 is true and t_2 can commit. Similarly t_1 is allowed to commit when t_2 commits first.

Thus, t_2 and t_1 can commit in any order. As far as the account object is concerned there is no conflict between t_1 and t_2 when they want to commit.

The use of private copies in dynamic validation scheme prohibits certain conflict serializable schedules. Under the circumstances (with private copies and distributed validation) the set of schedules allowed by dynamic validation is

better than similar schemes [Schw84, Herl87, Weih88]. This is because of the use of condition 4.4 instead of conditions 4.1 to 4.3. Two debit operations are not commutative according to conditions 4.1 to 4.3. This is true even if the arguments of the Debit operations are considered.

4.2.2. Cascaded Aborts

Since each transaction gets an execution copy and the commit process is arbitrated there are no cascaded aborts. Thus, schedules allowed by dynamic validation are recoverable.

4.2.3. Re-execution time

Two different empirical analyses are provided to compare the re-execution time with the execution time. The first analysis does not indicate the relative influence of the amount of nesting and the number of invocations on the ratio of the execution time to re-execution time. The second analysis indicates how the amount of nesting and number of invocations influence the ratio relative to each other.

Analysis 1

Let t_1 be a given transaction, n be the number of invocations in an action, and let p denote the average time taken by each invocation to return the response. Let q be the average time between receiving the response to one invocation and sending the next invocation. The following assumptions are made in this analysis:

1. Steps of an action are executed sequentially.
2. The average re-execution time of an action is equal to the average execution time.

3. Each action is performed on a distinct object.

Typically a transaction is a set of partially ordered actions. Thus, there may be one or more sequences of actions in a transaction. One such sequence may decide the execution time of the transaction. Due to the nesting among actions and message passing, the execution time of an action is expected to be larger than its re-execution time.

The total time to execute the actions on the average is denoted by *et* and is given by the equation 4.5.

$$et = (n - 1) * q + n * p \tag{4.5}$$

During the re-execution phase, the local steps of t_1 are executed in each participating object and the results of these local steps are compared with the results of the execution phase. Let m denote the time taken in an object to find whether the results are compatible. Since the re-execution is performed in parallel in all the participating objects and because of our assumptions 2 and 3 the total re-execution time is given by equation 4.6.

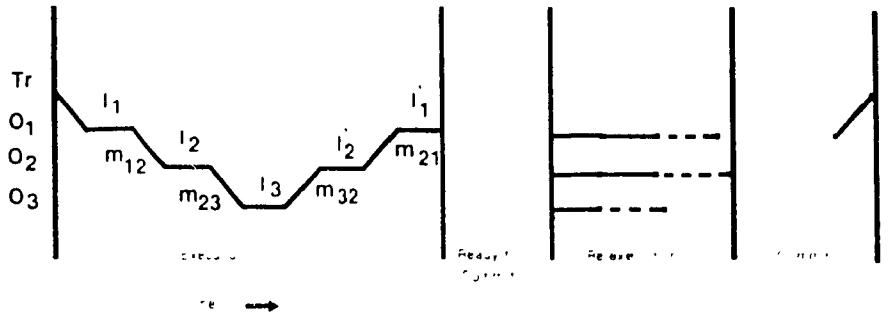
$$rt = p + m \tag{4.6}$$

In condition 4.1 two results are compared to find whether they are equal. Since local and comparison operations are performed on primitive objects (real, integer, etc) it is reasonable to assume that these operations are of equal complexity, and, thus, the re-execution time (rt) is given by the equation 4.6'.

$$rt = 2 * p \tag{4.6'}$$

This also suggests that the time taken to find if two results are compatible should be comparable to that of re-execution time else it could result in an inefficient

implementation of our scheme. One of our future goals is to identify compatibilities that can be verified efficiently. If we assume that an invocation of an action A on an object o_i results in invocation of action B on another object o_j then, since in the re-execution phase A and B will be executed in parallel, the average re-execution time $rt < 2 * p$. Thus, the larger the depth of an invocation the smaller the re-execution time. This is illustrated in Figure 4.1. Before the response is returned to the transaction, re-execution and subsequently validation are performed. If the validation succeeds the response is returned to the transaction.



l_i Local Steps
m_{ij} Invocation from o_i to o_j
r_{ij} response from o_i to o_j
dotted line - time taken to compare the results of local steps

Figure 4.1. Timing diagram of Execution and Re-execution for a Transaction with Three Participating Objects.

Except for a database of only primitive objects the depth l would be ≥ 2. In object-oriented systems the depth is usually much greater. For a depth of 2, $rt = p$. In most interactive and design environments the typical value of q is much greater than the typical value of p, thus,

$$et > 2 * n * p \qquad (4.5')$$

Thus the ratio of the execution time to re-execution time is at least $2n$:1 (from 4.5'). To sum up, the analysis is that the re-execution time will only be a small fraction of the execution time.

Analysis 2

The purpose of this analysis is to find out the relative effects of the number of invocations and the nesting among actions on the execution and re-execution time ratio. The following simplifying assumptions are made to get a concise expression for the ratio so that such a comparison is possible.

1. Steps of an action/transaction are executed sequentially.
2. All the leaf nodes of transaction execution tree are at level l.
3. Every node except the leaf level nodes contain n children.

Figure 4.2 represents the transaction execution tree of the assumed type with $n = 2$ and $l = 4$. Subscripted a represents an invocation.

Let c represent the time for a message to reach its destination and x represent the time taken by a local computation (execution of local steps). Let us also assume that each object performs some local computation before sending an invocation or response. Execution time et_k of an action at level k can be represented by the following recursive equation.

$$et_k = n * et_{k-1} + 2 * n * c + (n + 1) * x$$

Solution to this recursive equation is given by equation 1.7' if each leaf level node performs only one local computation.

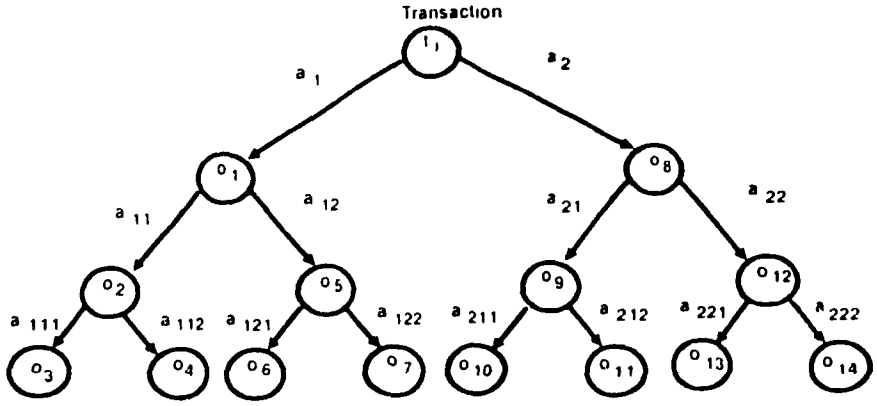


Figure 4.2. Transaction Execution tree of t_1.

$$et = n^{l-1}(x) + (2 * c \sum_{i=1}^{l-1} n^i) + ((n + 1) * x \sum_{j=1}^{l-1} n^j) \qquad 1.7'$$

The first term in the expression describes the time taken to execute the local steps at the leaf level of the tree. The second term gives the total time spent on communication. The third term gives the time spent on the local steps in non-leaf levels except the root. Equation 1.7' can be modified to include the think time q at the top level between two invocations of transactions. The last term in

equation 4.7 gives the time spent on thinking.

$$et' = n^{l-1}(x) + (2 * c\sum_{i=1}^{l-1} n^i) + ((n+1) * x\sum_{j=1}^{l-1} n^j) + (n-1) * q \qquad 4.7$$

The execution time of of Figure 4.2 can be computed by setting $n = 2$ and $l = 4$ in equation 4.7.

$$et = 29 * x + 28 * c + q$$

The re-execution time is computed as follows:

$$rt = (n+1) * x + m \qquad 4.8$$

Since each invocation in an object has $(n+1)$ local operations the first component of the re-execution is $(n+1) * x$; and the second component m is the average time taken to compare the results of the $(n+1)$ local computations. As previously discussed we can assume that m is of the order of $(n+1) * x$. Thus,

$$rt = 2 * (n+1) * x \qquad 4.9$$

For the transaction t_1, where $n = 2$, $rt = 6 * x$. Let ra represent the ratio of execution and re-execution time. For Figure 1.2

$$ra = (29 * x + 28 * c + q)/(6 * x) > 1.8 + 1.6\,(c/x)$$

In general, for large value of n and l

$$ra > (n^{l-1})/(2 * (n+1)) >> 1$$

4.2.4. Modularity

In most concurrency control schemes the conflict information is stored in a table, a *conflict table* [Schw84] or a *serial dependency relation* [Herl87]. When new

operations are introduced these tables are updated. This means, the person introducing the new operations must have complete knowledge of the other operations of the object. In the dynamic validation scheme there is no need for such knowledge; for each new operation only compatible results for the operation need to be introduced. Only knowledge of the operation being introduced is required.

4.2.5. Complexity of Dynamic Validation

In dynamic validation each object re-executes the local steps and decides whether to commit or restart a transaction. Thus, complexity of the dynamic validation can be computed in terms of local steps of actions. If an object were to re-execute n local steps and compare their results then the complexity of dynamic validation is $O(n)$ in local step. This is assuming the comparison of results or responses takes constant time.

4.3. Other Design Models from the Generic Model

The generic model of the *designactivity* presented in Chapter 3 is powerful enough to model all transactions or *designactivity* models described in Chapter 2 along with their correctness criteria. This is shown in four steps. In the first step, the list of constraints that give rise to serializable and recoverable schedules similar to [Schw84, Herl87, Herl87, Weih89] are presented. In the second step, serializability for interactive sequential transactions is presented. In the third step, the constraints that represent the nested transaction model with serializable schedules or application dependent correctness criteria [Bane85, Skar89] are discussed. Finally, some of the other possible *designactivity* models including [Lync84, Gare83, Gare87] are described.

I. Serializability for Sequential Transactions:

Validation constraints:

1. A project activity should contain only one node (a transaction).
2. A transaction should contain only a sequence of actions.
3. A transaction t_i should be allowed to commit only if no transaction t_j with serially dependent action has committed in any of the common participating objects after t_i has invoked an action on the object.

Unless stated otherwise the same set of validation and visibility constraints are associated with every *designactivity*. Condition 3 assumes the use of optimistic conflict serializable scheme with private copies. By replacing this constraint by the following, serializable schedules using static and dynamic two-phase locking schemes [Papa86] are obtained.

3′. A transaction t_i should not be allowed to execute a conflicting action if another live transaction t_j has already executed one of the conflicting actions in any of the common participating objects.

Serializability defined using equation 4.1 is weaker than the conflict serializability defined using equation 1.1 to 1.3. Unless or otherwise stated the term serializable refers to the one described by the definitions 3.11 and 3.13 using the equivalent of results described by compatible equation 1.1.

II. Interactive Sequential Transactions:

We can model the execution of interactive sequential transactions using equation 1.1. The use of equation 1.1 in condition 3 guarantees serializable schedules.

III. Nested Transactions:

If the following validation constraints and visibility constraints are applied

on our generic *designactivity* model, a *designactivity* model which is structurally similar to the nested transactions [Moss87] is obtained.

Validation Constraints:

1. A transaction contains only a sequence of actions.
2. A *designactivity* commits only when all its children are committed.
3. A transaction commits only when equation 4.1 is true for all its local steps.
4. A *designactivity* is aborted when its ancestor is aborted or if equation 4.1 is not true or due to some other reasons.

Visibility Constraints:

1. Private copies of a completed *designactivity* must be validated against and installed in favor of the private copies at its parent.
2. Private copies available at an ancestor are available to its descendents.

The validation and visibility constraints listed above create serializable schedules using dynamic validation. By replacing equation 4.1 in the validation constraints by equation 4.1' data types semantics are captured and added concurrency is possible. Most of the time a design is treated as complete when it satisfies certain design rules, functional specification, and performance specification. By replacing equation 4.1 by these specifications a correctness criterion that is application dependent [Bane85, Skar89] can be obtained. As long as a *designactivity* satisfies these requirements it is treated as complete. In this case the validation constraints associated with each transaction could be different. These constraints could even permit non-serializable schedules. With application dependent constraints and sequential

transactions, models like [Lync84, Garc83, Garc87] can be specified.

IV. Other Models:

The above visibility constraints can be altered easily to allow global commits. This will represent models that permit sharing across projects. A transaction can state the conditions under which it is using an object from other projects. For instance, a transaction can state that it makes use of the current version of an object and is not interested in the updates to this object. Similarly a transaction can state if it is to be aborted or just notified when a shared object is updated. How such a notification can be obtained with dynamic updates is discussed in the next section. This notification is applicable to design environment as well as traditional interactive environment. In traditional environment the notification will result in the restart of the notified transaction.

4.4. Applicability of Dynamic Validation to Design Activity

The dynamic validation scheme can be extended to the entire *designactivity* by treating each *designactivity* object as a transaction object and applying the same procedure.

In our validation scheme *designactivities* are validated only when the designer thinks it is complete. If they are invalid they are not aborted, instead, the designers are allowed to make the appropriate changes and attempt the commit iteratively. Since the validation is done at the end of a *designactivity* other *designactivities* need not be delayed. Result of the completed activity is made available to any other *designactivity* after establishing a dependency between the completed activity and the activity that makes use of its results. This allows *designactivities* to commit independently of their ancestors and making

them available globally.

In the backward dynamic validation scheme a transaction is validated only at the time of its commit. If the validation is not successful the transaction is asked to abort/rollback. Instead, if a live transaction is informed of a potential conflict due to the commit of other transaction then the transaction can take appropriate actions immediately, instead of finding out at the end that it has to abort/rollback. We describe a scheme using dynamic validation that detects potential conflicts.

4.4.1. Dynamic Backward Validation with Forward Check

In this scheme an extra phase is added to the commit. In this new phase, called the *forward checking* phase, the steps of the live transactions are re-executed on the state s'_{ij} obtained by re-executing the steps of the committing transaction. This is to say that each live transaction is validated against the state-to-be.

Transaction manager of t_1 sends commit-request message to all the participating object managers. The object manager of o_i acquires a lock on the object on behalf of t_1. Only a transaction with this lock can perform its commit-phase check in the object. This lock is released when the object receives the commit or abort message from the TM of t_1. It is necessary to hold a lock briefly at the commit stage to enforce common commit order among transactions. This lock will be of shorter duration compared to DBVL because in this scheme there may not be a re-execution of the committing transaction. After acquiring this lock local steps of the other live transactions are re-executed. Details of this procedure is explained in the following paragraphs.

Let $t_1, t_2, \ldots, t_n$ be the set of transactions executing in object o_i and let All

these transactions start with the same state s_i of o_i. Let $s_{i1}, s_{i2}, \ldots, s_{in}$ be the execution copies of the transactions t_1 to t_n respectively.

When a transaction t_1 wants to commit, it sends a request to o_i and other participating objects. Since no other transaction has committed in object o_i since t_1 started, O_i sends ready-to-commit message to the TM of t_1 and starts the re-execution of the local steps of every transaction t_k $(2 \leq k \leq n)$ on a copy of s_{i1} (state-to-be), s'_{ik}. Unlike the re-execution described in the last Chapter, this re-execution executes the local steps of the other live transaction in an object. To differentiate, this re-execution will be called *testing* or *forward checking*. The copy s'_{ik} is called the *testing copy* of the transaction t_k.

After an interval of time, O_i gets a message from the TM of t_1. If this message is an abort message, s_{i1} and all testing copies are deleted. The transactions t_2 to t_n continue their executions on their execution copies s_{i2} to s_{in} respectively, and t_1 is restarted.

If the message received from the TM of t_1 is a commit message then s_{i1} becomes the current state of the o_i.

The testing execution of of a transaction t_j $(2 \leq j \leq n)$ could be in one of the following four stages when O_i receives the message to commit.

1. The complete set of local steps from t_j received by O_i has been re-executed, and all these local steps are found to have the compatible results from s_{ij} and s'_{ij}. In this case the execution copy of t_j, s_{ij}, is discarded. Testing copy s'_{ij} is renamed as s_{ij} and becomes the new execution copy. Any new step from t_j to O_i will be executed on this s_{ij}. If the transaction t_j wants to commit at this stage, O_i can send ready-to-commit message without any delay because the local steps from t_j on the current state of o_i have already been performed.

2. Re-execution is terminated because the last local step has received incompatible result. In this case the transaction t_j is asked to restart by O_i and the private copies of t_j are discarded.

3. Only a proper subset of the local steps received up to that point of time has been re-executed and all these local steps have the same result from s_{ij} and s'_{ij}. In this case the testing is continued till one of the following conditions is true:

    a. A local step in testing receives incompatible result.

    b. No local step receives incompatible result.

    c. A transaction t_k $(k \neq j)$ commits in o_i.

If the testing stops due to condition (a) then the TM of t_j is asked to restart again. If the testing stops due to condition (b) then the execution copy of the transaction is discarded and the testing copy is made the new execution copy. Subsequent steps from t_j are executed on this new copy. It is also possible that t_j is the next transaction in o_i to start its commit, and when the testing of t_j is finished, TM of t_j is sent either a ready-to-commit message or a restart message. If it stops due to condition (c) then the testing copy is discarded and the execution continues with no changes. If $k = j$ then the re-execution of the t_j is continued after acquiring the lock. From now on, it is similar to the commit of t_j except that t_j still has to find out if it can commit in o_i.

Theorem 4.3: If *compatibility* is a transitive relation then schedules allowed by DBVFC are correct.

Proof: When a testing is performed for a transaction, it is checked whether its results are compatible with those that are obtained during the previous testing or

the execution. Only under this condition DCVFC replaces the private copies of the transaction by its testing copies. Since the *compatibility* is a transitive relation, results obtained during the commit phase are guaranteed to be compatible with the original execution phase results. □

In this scheme there are at most two private copies per transaction in an object and there is no race condition. The first part of the previous statement can be proved using the conditions under which testing copies are created and discarded. The second part of the statement follows from the fact that irrespective of the forward checking phase a transaction re-executes all its local steps on the current state before committing. Due to the forward checking phase the time complexity and space requirements of this scheme are usually more than that of dynamic validation.

One of the major disadvantages of the checking phase is the re-execution of steps of live transactions. In *designactivities* this is unlikely to be a major problem. It would be, however, beneficial to pinpoint any possibility of failure at an early stage. It should be noted that it is not necessary for an activity to abort even when the possibility of conflict exists.

The alternative to detection of possible conflicts using forward checking is to detect these possibilities using conflict tables. Conflict tables suffer from one major disadvantage: when a new operation is added to an object the conflict table has to be updated. These updates could be made only with a complete knowledge of other operations. In contrast, enhancing the compatible relation does not impose such restrictions.

The forward checking scheme not only restarts a transaction at an earlier stage, but also gives dynamically changing private copies [Maie86] to transactions. If the forward checking is successful for a live transaction then its private copies

are replaced with corresponding test copies without users' knowledge. The forward checking will reduce the number of transactions that are restarted. An alternative implementation of the scheme using timestamp and its correctness proof are discussed in [Goya88b].

Chapter 5

Conclusions

In this thesis we have attempted to address some of the drawbacks of traditional concurrency control schemes. Serializability as a correctness criterion has been implicit in these schemes. This correctness criterion is external to the application, domain or database behavior. Serializability is a major hindrance in the use of databases for long transactions. It leads to unavailability of data items for long periods or repeated aborts of long transactions [Papa86]. To eliminate the bottleneck created by the serializability we have proposed the use of application-, domain- or database- specific correctness criteria. These may be specified as a set of validation constraints. Validation constraints can be application dependent or independent. One way of specifying the constraints is by compatibility. This was demonstrated in Chapter 4.

We have demonstrated the use of our concurrency control scheme for design activities. These activities, usually, require sharing among designers. We have shown that sharing can be controlled by specifying a set of visibility constraints. Visibility constraints specify how objects are shared, copied etc. — including versions. The number of designs that fail validation can be reduced by notifying the *designactivities* that are affected by a committing design. Visibility constraints can be implemented using triggers.

We have presented a generic model of *designactivity*. Each node in the tree representation of the *designactivity* represents either a design activity or a transaction. We do not specify any constraints on the order of the actions contained in a transaction. This fits well with the object model where actions invoked on the same object can be executed concurrently when the actions do not

share common participating objects. Validation and visibility constraints are associated with each node.

The proposed concurrency control scheme is based on dynamic validation. In a dynamic validation scheme, transactions progress using private copies of objects. Only at commit time they are validated against the current copy of the object base. Performing the validation at the end of the *designactivities* is preferable for the following reasons: 1) it improves concurrency; 2) reduces delays between designactivities; and 3) validation constraints need not be specified until the end of the *designactivity*. We have also shown that in design environment the re-execution for validation takes only a small fraction of the total design time. An unsuccessful validation does not lead to an abort as in conventional schemes. Instead, it allows the designer to commit after making the appropriate changes.

Though the dynamic validation scheme has been developed for design applications, it can also be used for other interactive applications, such as groupware systems [Ellis89].

5.1. Implementation Issues

The proposed validation procedure requires that the commit or abnormal-event message be received from all the participating objects before deciding to commit or retry (after making changes). However, the *designactivity* object may not be aware of all the participating objects because of the object structure. This can be overcome in two ways:

1. Whenever an action of an object is executed on behalf of a design, the object identity is sent to the design and vice versa. The design object has the list of objects from which it must receive the commit or abnormal-event message.

2. Commit or abnormal event message is passed from the children to parent and vice versa.

If no insertions and updates of objects have been made, then the execution tree of a design will remain the same during the execution and commit phases. However, with insertions and updates the commit phase is slightly complicated. If we assume that the set of destination objects in a message is explicitly enumerated then the insertion of objects will not affect the re-execution. Deletion can be handled by retaining the OM of the deleted object till all the transactions that used the object are completed. Instead of enumeration, the destination objects can be specified using predicates. The effect of such specification on dynamic validation scheme needs further investigation.

Private copies of an object are created and destroyed by the object itself. An object creates a private copy only when it receives an invocation from another object. In other schemes distinct private or semi-private databases are created and managed by the global database manager.

We have used the decentralized control in dynamic validation scheme. There is no centralized object (scheduler) that controls the concurrent execution of transactions. Instead, participating objects and *designactivities* communicate among themselves to achieve the correct execution. The decentralized approach enables us to exploit the parallel execution at the object level.

In design applications there is little difference between the object creator and the user. In such an environment the users can be allowed to decide interactively about the validation constraints. For instance, one can decided whether two responses are compatible. This facility will allow designers to experiment with on-the-fly ideas.

5.1.3. Node Architecture

A successful implementation of the dynamic validation scheme can benefit from parallel execution, since checking the conditions can be done concurrently in every participating object. These conditions can be evaluated faster by allocating each object on a distinct processor element (PE). In practice such an allocation may not be possible and more than one object may be allocated to the same PE.

The following configuration of PEs will suit the above requirements. The PE that contains the node-manager object (NMO) that handles messages from other PEs. When a NMO receives a message for an object it checks if the object is dormant, if so the NMO sends a request to the secondary memory object manager to fetch the object. After moving the object from secondary storage to one of the processor element object manager (PEOM), the status of the object is changed to non-dormant. If the processor element containing the PEOM is free it is allocated to the object and the object becomes active. Otherwise it is kept in wait state till the processor becomes free. Once the object becomes active it can process the message that has been received by the NOM on its behalf. More than one object can be allocated to each PE.

When a NMO receives a message for an object, if the object is not dormant the message is sent to the PEOM that contains the active or the waiting object. The above approach is similar to the approach used in operating systems to handle processes; but unlike processes, objects are persistent entities. Thus, an object can be in dormant state, wait state or active state. The conditions under which state transitions take place are also different. Thus, techniques developed for processes can be used with appropriate changes for states and state transitions.

5.2. Future Research

One of the limitations of the proposed scheme is the difficulty in specifying the constraints. In dynamic validation some of these constraints are specified using compatible functions. To define compatibility functions, designers should have a good understanding of the application. We also need to study schemes for effective verification of compatibility using an efficient implementation of the concurrency control scheme.

When a transaction is found invalid the designer may have to undo the effect of some of the invocations. Due to the use of private copies the problem of undoing is different from the one presented in [Gare87]. Schemes that will make this process simple and efficient need to be further studied. A formal framework to present the ideas about validation, visibility constraint, and notification is also worth pursuing.

Shen [Shen88] specifies an algorithm for partially ordering a set of n concurrent operations under given constraints in such way that

- no operation can undo the effect created by some other operations(s) and
- no operation(s) block the execution of other operation(s).

In general this algorithm may need the enumeration of all $n!$ sequences of operations. However, for *weakly positive* constraints a polynomial time algorithm is possible. Feasibility of such a pessimistic algorithm for scheduling concurrent *designactivities* should be further investigated.

References

Ahls86. M. Ahlsen, A. Bjornerstedt, and C. Hulten, "OPAL: An Object-Based System for Application Development," *IEEE Data Engineering*, pp. 31-40, 1986.

Alme85. G.T. Almes, A.P. Black, E.D. Lazowska, and J.D. Noe, *The Eden System: A Technical Review, Vol SE-11, No 1*, pp. 43-59, Jan. 1985.

Ande86. T.L. Anderson, E.F. Eckland, Jr., and D. Maier, *PROTEUS Objectifying the DBMS User Interface*, 1986.

Atwo85. T.M. Atwood, "An Object Oriented DBMS for Design Support Application," in *Proc. IEEE COMPINT 85* , pp. 299-307, Montreal, Canada, 1985.

Badr87. B.R. Badrinath and K. Ramamritham, "Semantics-Based Concurrency Control: Beyond Commutativity," in *COINS Technical Report 86-18, Revised April '87* , University of Massachussets at Amherst, MA, April 1987.

Badr88. B.R. Badrinath and K. Ramamritham, "Synchronization of Transactions on Objects," in *IEEE Transactions on Computers Vol 37, No 5*, pp. 541-547, May 1988.

Banc85. F. Bancilhon, W. Kim, and H.F. Korth, "A Model of CAD Transactions," in *Proc. 10th International Conference on Very Large Data Bases*, pp. 25-33, Stockholm, Sept. 1985.

Banc87. J. Banerjee, W. Kim, H.J. Kim, and H.F. Korth, "Semantics and Implementation of Schema Evolution in Object Oriented Databases," in *Proc ACM SIGMOD Conference on Management of Data*, San Francisco, CA, May 1987.

Beec87. D. Beech, "Groundwork for an Object Database Model," in *Research Directions in Object-Oriented Programming, eds. B. Shriver and P Wegner*, pp. 317-354, The MIT Press, Cambridge, MA , 1987.

Beer89. C. Beeri, P.A. Bernstein, and N. Goodman, "A Model for Concurrency in Nested Transaction Systems," in *Journal of ACM, Vol 36, No 2* , pp. 230-269, April, 1989.

Bern81. P.A. Bernstein and N. Goodman, "Concurrency Control in Distributed Database Systems," in *ACM Computing Surveys, Vol. 13, No 2*, pp. 185-221, June 1981.

Bern87. P.A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison Wesley, Reading, MA, 1987.

Blac85. A.P. Black, "Supporting Distributed Applications: Experience with Eden, Vol. SE-11, No. 12," *IEEE Tran on Software Engineering*, pp. 181-193, Dec. 1985.

Booc86. G. Booch, "Object-Oriented Development," *IEEE Tran On Software Engineering, Vol SE-12, No 2*, pp. 211-221, Feb. 1986.

Buch86. A.P. Buchmann, R.S. Carrera, and M.A. Vazquez-Galindo, "CAD-DBMS," in *International Workshop on Object-Oriented Database Systems '86*, pp. 2-6, CA, Sept. 1986.

Bune79. O.P. Buneman and E.K. Clemons, "Efficiently Monitoring Relational Databases," in *ACM Transaction on Database Systems, Vol 4, No 3* , pp. 368-382, Sept. 1979.

Care86. M.J. Carey, D.J. DeWitt, D. Frank, G. Graefe, M. Muralikrishna, J.E. Richardson, and E.J. Shekita, "The Architecture of the EXODUS Extensible DBMS," in *Proc International Workshop on*

Object-Oriented Database Systems, pp. 52-53, Sept. 1986.

Chou86. H.T. Chou and W.Kim, "A Unifying Framework for Versions of Schema in CAD Environment," in *Proc 11th International Conference on Very Large Data Bases*, Kyoto, Japan, Aug. 1986.

Chou88. H.T. Chou and W. Kim, "Versions and Change Notification in an Object-Oriented Database System," in *Proc 25th Design Automation Conference*, June 1988.

Dall85. W. Dally, *VLSI Architectures for Concurrent Data Structures*, Ph.D. dissertation, Dept. of Computer Sc., California Institute of Technology, CA, 1985.

Dasg86. P. Dasgupta and M. Morsi, "An Object-Based Distributed Database System Supported on the Clouds OS," in *Technical Report GIT-ICS-86/07*, pp. 1-21, Georgia Institute of Technology, GA, July 1986.

Ditt85. K. Dittrich and R. Lorie , "Version Support for Engineering Database Systems," in *IBM Research Report RJ4769, IBM Research*, July 1985.

Ditt86. K.R. Dittrich, "Object-Oriented Database Systems: The Notion and the Issues," in *International Workshop on Object-Oriented Database Systems '86*, pp. 2-6, California, Sept. 1986.

Elli89. C.A. Ellis and S.J. Gibbs, "Concurrency Control in Groupware Systems," in *Proc of ACM SIGMOD International Conference on Management of Data*, Portland, OR, June 1989.

Eswa76. K.P. Eswaran, J.N. Gray, R.A. Lorie, and I.L. Traiger, "The Notions of Consistency and Predicate Locks in a Database System," in *Communications of the ACM Vol 19 No 11*, pp. 624-633, Nov. 1976.

Cohe86. M.J. Jackson, *The Specification of Complex Systems*, Addison Wessley, Reading, MA, 1986.

Fern89. M.F. Fernandez and S.B. Zdonik, "Transaction Groups: A Model for Controlling Cooperative Transactions," in *3rd International Workshop on Persistent Object Systems*, Queensland, Australia, Jan. 1989 .

Feke87. A. Fekete, N. Lynch, M. Merrit, and W. Weihl, "Nested Transactions and Read/Write Locking," in *6th ACM Symposium on Principles of Database Systems*, pp. 97-111, 1987 .

Gare83. H. Garcia-Molina, "Distributed Database," in *ACM Trans. on Database Systems, Vol 8, No 2, June 1983.*, pp. 186-215, June 1983..

Gare87. H. Garcia-Molina and K. Salem, "Sagas," in *Proc ACM SIGMOD International Conference on Management of Data*, pp. 249-259, May, 1987.

Gawl85. D. Gawlick, "Processing "hot spots" in High Performance Systems," in *Proceedings of Spring COMPCON '85*, pp. 249-252, 30th IEEE Computer Society International Conference, San Francisco, CA, 1985.

Gold83. A. Goldberg and D. Robson, *Smalltalk-80, Language and its Implementation.* , Addison Wesley, Reading, MA, 1983.

Goya87. P. Goyal, T.S. Narayanan, Y.S. Qu, and F. Sadri, "Requirements for an Object Based Integrated Systems Environment," in *Technical Report CSD-87-07*, Concordia University, Montreal, Sept. 1987.

Goya88a. P. Goyal, T.S. Narayanan, and F. Sadri, "Characterization of Design Application Transactions for Concurrency Control," in *Technical Report CSD-88-04*, Concordia University, Montreal, Canada, Aug. 1988.

Goya88b. P. Goyal, T.S. Narayanan, and F. Sadri, "Concurrency Control for Design Object Bases," in *Technical Report No. CSD-88-05*, Concordia University, Montreal, Canada, 1988.

Gray78. J.N. Gray, "Notes on Database Operating System," in *Lecture Notes in Computer Science, Vol 60, eds R. Bayer et Al*, pp. 393-481, Berlin: Springer-Verlag, 1978.

Hadz85. T. Hadzilacos and C. Papadimitriou, "Algorithmic Aspects of Multiversion Concurrency Control," in *4th ACM Symposium on Principles of Database Systems*, pp. 96-104 , 1985.

Hadz88a. T. Hadzilacos, "Serialization Graph Algorithms For Multiversion Concurrency Control," in *ACM PODS '88*, pp. 193-200, Austin, Texas, March 1988.

Hadz88b. T. Hadzilacos and V. Hadzilacos, "Transaction Synchronization in Object Bases," in *ACM PODS '88*, pp. 193-200, Austin, Texas, March 1988.

Hard86. T. Harder, "New Approaches to Object Processing in Engineering Databases," in *Proc International Workshop on Object-Oriented Database Systems*, p. 217-222, Sept. 1986.

Hask82. R.L. Haskin and R.A. Lorie, "On Extending the Functions of a Relational Database System," in *Proc ACM SIGMOD International Conference on Management of Data*, pp. 207-212, 1982.

Herl87. M.P. Herlihy, "Optimistic Concurrency Control for Abstract Data Types," in *ACM Operating Systems Review, Vol 21, No 2*, pp. 33-44, April 1987.

Herl88. M.P. Herlihy and W.E. Weihl, "Hybrid Concurrency Control for Abstract Data Types," in *ACM PODS '88*, pp. 201-210, Austin,

Texas, March 1988.

Katz83. R.H. Katz and S. Weiss, "Transaction Management for Design Databases," in *Working Paper*, 1983.

Katz84. R. Katz and T. Lehman, "Database Support for Versions and Alternative of Large Design Files," *IEEE Transaction on Software Engineering , Vol SE-10, No. 2*, pp. 191-200, March 1984.

Katz86. R. Katz, E. Chang , and R. Bhateja, "Version Modeling Concepts for Computer-Aided Design Databases," in *Proc. ACM SIGMOD International Conference on Management of Data*, pp. 379-386, Washington, D.C., May 1986.

Keta86. M.A. Ketabchi, "Object-Oriented Data Models and Management of CAD databases," in *Proc. International Workshop on Object-Oriented Database Systems*, pp. 223-224, Sept. 1986.

Kim84. W. Kim, R. Lorie, D. McNabb, and W. Plouffe, "A Transaction Mechanism for Engineering Design Databases," in *Proc. 9th International Conference on Very Large Data Bases*, pp. 355-362, Singapore, Aug. 1984.

Kim88. W. Kim and H.T. Chou, "Versions of Schema for Object-Oriented Databases," in *Proc 14th International Conference on Very Large Data Bases*, pp. 148-159, Los Angeles, CA, Sept. 1988.

Kohl81. W. Kohler, "Decentralized Computer Systems," in *ACM Computing Survey, Vol 13 No 2*, pp. 149-183, 1981.

Kort83. H.F. Korth, "Locking Primitives in a Database Systems," in *Journal of the ACM, Vol 30, No 1*, pp. 55-797, Jan. 1983.

Kort88a. H.F. Korth, W. Kim, and F. Bancilhon , "On Long Duration CAD Transactions," in *Information Sciences, 46*, pp. 73-107, 1988.

Kort88b. H.F. Korth and W. Kim, "A Concurrency Control Scheme for CAD Transactions," in *Technical Report*, University of Texas, Austin, Texas, April 1988.

Kort88c. H.F. Korth, and G. Speegle, "Formal Model of Correctness without Serializability," in *ACM SIGMOD International Conference on Management of Data*, pp. 379-386, 1988.

Kung81. H.T. Kung and J.T. Robinson, "On Optimistic Methods for Concurrency Control," in *ACM Transactions on Database Systems, Vol. 6, No 2*, pp. 213-226, June 1981.

Lieb81. H. Lieberman, *A Preview of Act-1, MIT A I Memo, No 625*, 1981.

Lieb86. H. Lieberman, "Using Prototypical Objects to Implement Shared Behavior in Object Oriented systems," in *OOPSLA Proceedings '86*, pp. 214-223, Sept. 1986.

Lori83. R. Lorie and W. Plouffe, "Complex Objects and Their Use in Design Transactions," in *Proc Databases for Engineering Applications, Database Week 1983(ACM)*, pp. 115-121, May 1983.

Lync84. N.A. Lynch, "Multilevel Atomicity - A New Correctness Criterion for Database Concurrency Control," in *ACM Transactions on Database Systems, Vol 8, No 4*, pp. 484-502, Dec. 1984.

Maie86. D. Maier, P. Nordquist, and M. Grossman, "Displaying Database Objects," *First International Conference on Expert Database Systems*, Charleston, South Carolina, April 1986.

Mano86. F. Manola and U. Dayal, "PDM: An Object-Oriented Data Model," in *Proc International Workshop on Object-Oriented Database Systems*, pp. 18-25, Sept. 1986.

Moss87. J.E.B. Moss, "Nested Transactions: An Introduction," in *Concurrency Control and Reliability in Distributed Systems, ed. B. Bhargava*, pp. 395-425, Von Nostrand Reinhold Company, NY, 1987.

ONei86. P.E. O'Neil, "The Escrow Transactional Method," in *ACM Transactions on Database Systems*, pp. 405-430, Dec. 1986.

Papa86. C. Papadimitriou, *The Theory of Database Concurrency Control*, Computer Science Press, Rockville, MD, 1986.

Pu88. C. Pu, G.E. Kaiser, and N. Hutchinson, "Split-Transactions for Open-Ended Activities," in *Proc. 14th International Conference on Very Large Data Bases*, pp. 26-36, Los Angeles, CA, Sept. 1988.

Rao86. K.V.B. Rao, "An Object-Oriented Framework for Modeling Design Data," in *Proc. International Workshop on Object-Oriented Database Systems*, pp. 53-59, Sept. 1986.

Reut82. A. Reuter, "Concurrency on High-traffic Data Elements," in *ACM Symposium on Principles of Database Systems*, pp. 83-92, NY, Mar. 1982.

Scha78. T. Schaefer, "The Complexity of Satisfiability Problems," in *ACM Symposium on Theoretical Computing*, pp. 216-226, 1978.

Schw84. P.M. Schwarz and A.Z. Spector, "Synchronizing Shared Abstract Data Types," in *ACM Transactions on Computer Systems, Vol. 2, No 3*, pp. 223-250, Aug. 1984.

Schw86. P. Schwarz, W. Chang, J.C. Freytag, G. Lohman, J. McPherson, C. Mohan, and H. Pirahesh, "Extensibility in the Starburst Database System," in *International Workshop on Object-Oriented Database Systems*, pp. 85-94, CA, Sept. 1986.

Sheu88. P.C. Sheu, "Operation Management in Object-Oriented Knowledge Bases," in *International Journal of Intelligent Systems, Vol 3*, pp. 381-397, 1988.

Shib87. E. Shibayama and A. Yonezawa, "Distributed Computing in ABCL/1," in *Object-Oriented Concurrent Programming, eds A Yonezawa and M. Tokoro*, pp. 92-128, The MIT Press, Cambridge, MA , 1987.

Skar86. A.H. Skarra and S. Zdonik , "The Management of Changing Types in an Object-Oriented Database," in *Proc. Object-Oriented Programming Systems, Language, and Applications*, Portland, OR, Oct. 1986.

Skar86. A.H. Skarra, "Concurrency Control for Cooperating Transactions in an Object-Oriented Database," in *ACM Workshop on Object-Based Concurrent Systems, SIGPLAN Notices Spring '89*, 1989 .

Spoo86. D.L. Spooner, "An Object-Oriented Data Management System for Mechanical CAD," in *International Workshop on Object-Oriented Database Systems*, pp. 233-234, CA, Sept. 1986.

Stei87. L.A. Stein, "Delegation Is Inheritance," in *OOPSLA '87 Proceedings*, pp. 138-146, Orlando, FL, Oct. 1987.

Ston86a. M. Stonebraker and L.A. Rowe, "The Design of POSTGRES," in *Proc. ACM/SIGMOD International Conference on Management of Data*, pp. 340-355, Washington, DC, May 1986.

Ston86b. M.R. Stonebraker, "Object Management in POSTGRES using Procedures," in *Proc. International Workshop on Object-Oriented Database Systems*, pp. 66-72, Sept. 1986.

Unga87. D. Ungar and R.B. Smith, "SELF: The Power of Simplicity," in *OOPSLA '87 Proceedings*, pp. 227-240, Orlando, FL, Oct. 1987.

Walp88. J. Walpole G.S. Blair, J. Malik, and J.R. Nicol, "A Unifying Model For Consistent Distributed Software Development," in *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symp. on Practical Software Development Environments*, pp. 183-190, Boston, MA, Nov. 1988.

Weih89. W.E. Weihl, "Local Atomicity Properties: Modular Concurrency Control for Abstract Data Types," in *ACM Transactions on Programming Languages and Systems Vol 11, No 2*, pp. 249-282, April 1989.

Yoko85. Y. Yokote and M. Tokoro, "Concurrent Smalltalk - An Object Oriented Concurrent Programming Language ," in *Object-Orientation - Tutorials and Selected Papers from the Workshop on Object Concurrent Computing, ed A Suzuki, Kyoritsu Pub , in Japanese* , 1985.

Yoko87. Y. Yokote and M. Tokoro, "Concurrent Programming in Concurrent Smalltalk," in *Object-Oriented Concurrent Programming, eds A Yonezawa and M Tokoro*, pp. 129-158, The MIT Press, Cambridge, MA , 1987.

Yone87. A. Yonezawa, E. Shibayama, T. Takada, and Y. Honda, "Modeling and Programming in an Object-Oriented Concurrent Language ABCL/1," in *Object-Oriented Concurrent Programming, eds A. Yonezawa and M Tokoro*, pp. 55-89 , The MIT Press, Cambridge, MA , 1987.

Zdon86. S. Zdonik, "Object Management Systems for Design Environments,"

in *Proc International Workshop on Object-Oriented Database Systems*, pp. 23-29, 1986.