Transaction and data models for design databases

Muruganandan Kumar

Iowa State University
INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI
University Microfilms International
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
313/761-4700  800/521-0600
Transaction and data models for design databases

Kumar, Muruganandan, Ph.D.

Iowa State University, 1991
Transaction and data models for design databases

by

Muruganandan Kumar

A Dissertation Submitted to the
Graduate Faculty in Partial Fulfillment of the
Requirements for the Degree of
DOCTOR OF PHILOSOPHY

Major: Computer Science

Iowa State University
Ames, Iowa
1991
To my parents:

Kumarasamy and Komalavalli

Narayanaswamy and Janaki.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>SECTION</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACKNOWLEDGEMENTS</td>
<td>ix</td>
</tr>
<tr>
<td>INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>DATA MODELS</td>
<td>4</td>
</tr>
<tr>
<td>Semantic Data Models</td>
<td>4</td>
</tr>
<tr>
<td>Geometric Data Models</td>
<td>6</td>
</tr>
<tr>
<td>Object-Oriented Models</td>
<td>7</td>
</tr>
<tr>
<td>TRANSACTION MODELS</td>
<td>10</td>
</tr>
<tr>
<td>Transactions in Design Databases</td>
<td>11</td>
</tr>
<tr>
<td>ROAD MAP</td>
<td>12</td>
</tr>
<tr>
<td>PART I. ISSUES IN DESIGN DATABASES</td>
<td>15</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>16</td>
</tr>
<tr>
<td>INTRODUCTION</td>
<td>17</td>
</tr>
<tr>
<td>ARCHITECTURE</td>
<td>19</td>
</tr>
<tr>
<td>TRANSACTION MODELS</td>
<td>21</td>
</tr>
<tr>
<td>Classical Transaction Model</td>
<td>21</td>
</tr>
<tr>
<td>Nested Transactions</td>
<td>24</td>
</tr>
<tr>
<td>DATA MODELS</td>
<td>26</td>
</tr>
</tbody>
</table>
PART II. TRANSACTION MODELS FOR DESIGN ENVIRONMENTS

ABSTRACT .......................... 41
INTRODUCTION ........................... 42
DESIGN ENVIRONMENT .................... 44
UNIFORM TRANSACTION MODEL ............. 48
  Multiple Versions ...................... 50
  Coherent Schedules .................... 51
  Object Consistency .................... 53
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>PART III. A DATA MODEL FOR DESIGN OBJECTS</td>
<td>71</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>72</td>
</tr>
<tr>
<td>INTRODUCTION</td>
<td>73</td>
</tr>
<tr>
<td>REQUIREMENTS OF A DESIGN DATA MODEL</td>
<td>76</td>
</tr>
<tr>
<td>COMPLEX OBJECTS</td>
<td>78</td>
</tr>
<tr>
<td>INTERFACE AND IMPLEMENTATION</td>
<td>82</td>
</tr>
<tr>
<td>Hierarchy of Interfaces</td>
<td>83</td>
</tr>
<tr>
<td>Attribute Types</td>
<td>86</td>
</tr>
<tr>
<td>INSTANTIATION</td>
<td>87</td>
</tr>
<tr>
<td>Parametric Instantiation</td>
<td>90</td>
</tr>
<tr>
<td>Conditional Instantiation and Lazy Evaluation</td>
<td>92</td>
</tr>
<tr>
<td>OPERATIONS ON COMPLEX OBJECTS</td>
<td>96</td>
</tr>
<tr>
<td>Operations on Object Interfaces</td>
<td>96</td>
</tr>
<tr>
<td>Operations on Object Versions</td>
<td>97</td>
</tr>
<tr>
<td>Operations on Instances</td>
<td>97</td>
</tr>
<tr>
<td>CONCLUSION</td>
<td>99</td>
</tr>
</tbody>
</table>
PART IV. CONCURRENCY CONTROL IN DESIGN DATABASES

ABSTRACT ......................................................... 101
INTRODUCTION ..................................................... 102
SYSTEM ARCHITECTURE ......................................... 103
DESIGN OBJECT MODEL ........................................... 106
   Object Interface and Implementation ......................... 107
   Instances ..................................................... 108
   Composite Objects ........................................... 110
OPERATIONS ON COMPLEX OBJECTS .............................. 115
CONCURRENCY CONTROL ........................................... 117
   Lock Protocol ................................................ 117
   Locking Modes ............................................... 118
   Lock Upgrade ............................................... 123
   Operations on Composite Objects ......................... 125
   Atomic Subtrees (AST) ....................................... 127
   Protocol for Accessing Composite Objects .................. 129
HYPOTHETICAL TRANSACTIONS .................................. 130
   Hypothetical Locks ......................................... 130
   Lock Upgrade ............................................... 131
DEADLOCK AND RECOVERY ...................................... 134
CONCLUSION ....................................................... 136
# PART V. TRANSACTION MANAGEMENT IN DESIGN

## DATABASES

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTRACT</td>
<td>138</td>
</tr>
<tr>
<td>INTRODUCTION</td>
<td>139</td>
</tr>
<tr>
<td>THE MODEL</td>
<td>142</td>
</tr>
<tr>
<td>Derivation Graph and Design Scheme</td>
<td>142</td>
</tr>
<tr>
<td>Design Objects</td>
<td>143</td>
</tr>
<tr>
<td>DATABASE AND CONSISTENCY</td>
<td>145</td>
</tr>
<tr>
<td>Serializability</td>
<td>145</td>
</tr>
<tr>
<td>Consistency</td>
<td>147</td>
</tr>
<tr>
<td>NESTED DESIGN TRANSACTIONS</td>
<td>150</td>
</tr>
<tr>
<td>Nested Transactions</td>
<td>150</td>
</tr>
<tr>
<td>Multiple Commit Points</td>
<td>153</td>
</tr>
<tr>
<td>Correctness</td>
<td>154</td>
</tr>
<tr>
<td>COOPERATING AND SUBCONTRACTOR TRANSACTIONS</td>
<td>157</td>
</tr>
<tr>
<td>Conversation Interface</td>
<td>158</td>
</tr>
<tr>
<td>Recovery and Backout Sphere</td>
<td>159</td>
</tr>
<tr>
<td>Implementation</td>
<td>161</td>
</tr>
<tr>
<td>Subcontractor Transactions</td>
<td>165</td>
</tr>
<tr>
<td>OPERATIONS</td>
<td>167</td>
</tr>
<tr>
<td>CONCLUSION</td>
<td>170</td>
</tr>
</tbody>
</table>
ACKNOWLEDGEMENTS

I am forever indebted to my advisor Dr. Johnny Wong for things too numerous to list here. But for his constant help and support, this work would not have been possible.

I would like to thank Dr. Oldehoeft for kindling my interest in Operating Systems, and also for his help and support. I express my gratitude to Dr. Kothari for his advice and guidance, throughout my stay at Iowa State.

I thank my committee members Dr. Miller, Dr. Molian and Dr. Davis not only for their time, but also for the suggestions they gave to this dissertation.

I would also like to acknowledge Pinaki Shah and Linda Null for their many helpful suggestions.

Many thanks are due to my friends, particularly, Thyagu, Ramu and Srikanth for their support and for making life in Ames pleasant.

Most importantly, I wish to express my gratitude to my parents, my brother, and my sisters, for their unwavering faith and constant encouragement, and to my wife Priya, for enduring me all these years and for being my most valuable source of support.
INTRODUCTION

Advances in VLSI technology, computer architecture and the availability of inexpensive memory chips have resulted in development of powerful but relatively inexpensive computers. This phenomenon, coupled with the advances in computer networking, has fundamentally changed the style of computing. Interconnected workstations have replaced the terminals that were tied to the central main-frame computers. Distributed computing is becoming the norm rather than the exception.

Consequently, computers are being used in nontraditional tasks such as computer aided design (CAD), office automation, and software development, to name but a few. In these applications, computers are primarily used to manage the large volume of data that are generated, and to coordinate access to the data. The database associated with these nonconventional applications will be henceforth referred to as design databases. Conventional general-purpose database management systems were developed mainly to manage data and access to them under transaction mechanisms. Transactions are logically related set of operations that are used to query and modify the database state. Transactions, as popularized by Gray in [34] have three important properties that make them ideally suited for accessing and manipulating long-lived data. By invoking a transaction, users are assured that each transaction will:
1. execute exactly once (reliability). 

2. maintain consistency constraints even in the presence of temporary violations that might arise due to other concurrently executing transactions (consistency). 

3. change the state permanently, once the transaction commits, even in the presence of failures (permanence). 

It is evident that these features of transaction mechanism can be exploited while building a resilient system for advanced applications such as those pointed out earlier. Design processes can be modeled as a set of transactions, accessing design data that is shared by the different designers using the system. However, the conventional database and transaction mechanisms, cannot be directly applied to manage data in design environments. The wide spectrum of applications, the nature of the data, and the long duration transactions encountered are the main impediments in applying the conventional model to design databases.

The conventional transaction mechanism is based on view-serializability: only view-serializable schedules of transactions are acceptable. View-serializable executions are those in which the execution effect of a set of transactions is identical to that of a serial execution of the same set of transactions in some serial order. However, view-serializability has been shown to be NP-complete [64] and is too involved to be the basis for an on-line scheduling mechanism. In contrast, another notion of serializability – conflict-serializability has been proposed, and is in fact the first notion of correctness to be proposed for concurrency control [27]. While several mechanisms have been proposed to allow only serializable schedule [18], two-phase locking
proposed by Eswaran and others [27] has been the most widely studied and used. It has been shown that in the absence of other semantic information, two-phase locking is optimal [83]. In design environments, view-serializability or conflict-serializability requirement is unnecessarily restrictive. In addition, there are other consistency requirements that have to be considered. We formalize these consistency requirements and propose mechanisms to maintain the consistency criteria. More detailed survey of transactions and serializability issues are presented later in this section.

One of the major issues that need to be resolved in CAD databases and other databases used in advanced applications is a model for representing the design objects. The conventional data models such as the relational model assumes that the data are arranged in a set of files consisting of a sequence of homogeneous records. However, in the non-classical applications that are being considered here, data objects are complex and cannot be easily represented in a small number of homogeneous records. A successful model for the design transactions has to address the issue of data modeling.

In the subsequent sections we present a brief survey of the three concepts that are related to design databases - data models, the concept of transactions, and transaction mechanisms suitable for design environments.
DATA MODELS

Although commercial database systems based on data models tied to physical data organization such as hierarchical and network databases have been available for over two decades, it is after the introduction of the relational model [21] the area has witnessed tremendous growth. The relational model provided data independence to the designers and users of database. The model facilitated the users to abstract out the physical organization of data and provided a conceptual level for mapping the external world to the internal representation while providing an algebra (non-procedural) for manipulating the data. Design applications which include CAD/CAM, VLSI design and CASE generate and manipulate large volume of data which can benefit from a database management system. However, as pointed out earlier, the relational model does not offer sufficiently rich conceptual model for non-traditional applications. Implementors of special-purpose systems involving large volume of data have been using customized file systems rather than DBMS to manage their data. As the scope and the number of the non-traditional applications increase, the need for providing uniform and standard data models (such as EDIF for VLSI) design becomes paramount. In addition, DBMS systems and data-models are needed to support such applications. A brief survey of data models that have proposed to extend the traditional models (relational, hierarchical, and network) are now presented.

Semantic Data Models

Semantic models were introduced primarily as schema design tools. The initial emphasis was to model the relationships in database applications accurately [19, 73].
Semantic models provided a higher level of abstraction for modeling data. Semantically based data models provide the following advantages over traditional systems [38]:

1. Convenient abstraction mechanisms for modeling (so that conceptual model is closer to external model).

2. Economy of expression as the data model itself has semantics embedded within it allowing for greater ease of use.

3. Increased separation of conceptual and physical components of the database modeling levels.

4. Integrity maintenance without the explicit use of navigation or intra-object connections at the physical level.

5. Decreased semantic overloading of relationship types.

Semantic data models provide convenient mechanisms to allow database specifications to evolve incrementally. This is because, semantic models provide top-down schema design. Most models provide mechanisms to represent atomic types and also constructor types. The two most common constructors that are widely supported are aggregation also called relationship in ER model ([19]) and generalization also called as association. An aggregation is a composite object (e.g., address) composed of other objects defined earlier (e.g., ordered list of: Street, City, Zipcode). Generalization construct is used to represent sets of objects of the same type (e.g., a relation Persons where elements are instances of different Person objects). While aggregation and generalization are the most common constructors, sets, vectors, ordered sets,
and matrices are also supported by models such as the semantic association model (SAM* [74]). One of the main drawbacks of the relational model and other models based on it is that the data objects (tuples) should be composed of atomic attributes. This requirement is known as first normal form (INF). Extensions to the relational model generally relax the requirement, resulting in models based on non-first normal forms (NF^2) [1].

In this work, we consider an extension of the ER model proposed by Chen [19]. The model extends the traditional models and incorporates semantic information. In ER model, the two primary constructs are the entity and relationships. The model explicitly supports one-to-one, one-to-many, and many-to-many relationships. Insertion and deletion constraints can be specified by existence dependencies. Several abstraction mechanisms have been added to the model to extend it [11]. An extensive survey on semantic data models and a comparative study of various models are presented in [38] and [65].

Geometric Data Models

Recent surge of interest in CAD/CAM and the use of robots in manufacturing and mechanical assembly have necessitated the development of models suitable for representing physical and geometric properties of real objects. Currently, all commercial CAD/CAM applications are based on customized file system, rather than a database management system. This, again, is due to the limitations of current database management systems in efficiently managing the data encountered in such applications. In addition, the users of CAD/CAM systems - engineers - are not nec-
necessarily database experts, to custom design their database schemas, based on the (database) design constraints (the normal forms 1NF, 3NF etc..)[77]. Several models have now been developed for representing engineering data.

Database support is feasible for three representation schemes for solid objects [59]:

1. Constructive solid geometry (CSG), which is a volumetric representation of geometric parts in which an object is described as a composition of a few primitive parts with motional or combinatorial operators. This is the most widely used representation in CAD/CAM.

2. Boundary representation (BR) in which an object is represented as a set of non-overlapping faces. Each face is modeled by its bounding edges and vertices. A CSG representation can be readily converted into the BR form.

3. Primitive instancing, where each object type is defined as a special instance of a generic primitive object.

In addition to rapid retrieval of data, the model should facilitate application of operations encountered in design such as translation, scaling and rotation, on the data. Another important consideration is the ability to represent multiple objects. An analysis of the models suggested for geometric modeling can be found in [41].

Object-Oriented Models

Object-oriented databases have been proposed by many researchers as a solution to most of the problems encountered in supporting advanced applications (e.g., see
Two kinds of object-orientation have to be distinguished: (1) the structural approach such as the 'complex object' in [56] and the molecule paradigm, suggested in [10, 11]; (2) the behavioral approach that has its origins from programming languages. Here, we will refer only the behavioral models as 'object-oriented'.

While there has been considerable debate about what features should be embodied by an object-oriented model and the database [4, 23], the issues and the core concepts of object-orientation have been well established [6, 43, 44]. The core concepts in object-oriented data models are:

1. Object and Identifier: Any real-world entity is modeled as an object uniformly with its unique identifier which is independent of the values of the object attributes.

2. Attributes and Methods: The state of the object is captured by its attributes which are objects in their own right. The behavior of the object is captured by methods (program code) which operate on the attributes. The methods can be invoked only by sending messages to the object.

3. Class: A class is the means of grouping all objects that have the same set of attributes and methods. An object can belong to only one class; members of the class are related to the class by 'is-an-instance-of' relation.

4. Class Hierarchy and Inheritance: A new object class can be derived from an existing class. The new class is called the subclass of the existing class which is now the superclass. The new class inherits all its attributes and methods from its superclass and these may be augmented or modified in the subclass. The subclass-superclass induces a hierarchy called the class hierarchy.
Data modeling issues pertaining to object-oriented applications are discussed in [9, 42]. In recent years, several object-oriented databases have been built with features to support design databases (see [46] for a case study and implementation issues). However the following issues remain, some of which are inherent to the paradigm and are not expected to be solved easily with progress in the field:

1. Lack of common data model.
2. Lack of formal model such as the relational model.
3. A query language that will exploit the full power of the object-oriented system may be computationally unsafe.
4. Access methods are not sufficiently fast compared to conventional systems.
5. Schema changes and change propagation issues are (if) resolved in an ad-hoc manner.

However, it is now becoming increasingly clear that as the technology matures, object-oriented systems will be seen as attractive solutions to special-purpose applications.
TRANSACTION MODELS

Atomic action or transaction is a sequence of logically related operations on shared data that preserve the consistency of the data even in the presence of concurrency and failures. The two main issues that concern transaction mechanisms, other than practical considerations such as fast storage systems, secondary data management etc., are concurrency control and recovery. If it is assumed that in the absence of failures and concurrency, each process when acting alone preserves the consistency of the system, then it follows that if the effect of a set of transactions $T$ executing concurrently resulting in a schedule $S$ is equivalent (in some aspect) to some serial execution $S'$, of the same set of transactions $T$, then the schedule $S$ is correct.

There are several notions of correctness. For conventional systems (including tracking databases), view serializability is considered as the appropriate criteria [27, 62]. Of the other criterias, state serializability and co-serializability are of interest to us. A discussion of these can be found in [78]. Theoretical aspects of the different models can be found in [64, 15, 63, 17]. While two-phase locking ([27]) is known to be the first protocol for generating serializable schedules, in recent years several proposals have been made [18, 14].

Most proposals for concurrency control use the syntax of the primitive operations of transactions (e.g., read and write operations) rather than the semantics of the operations. To increase the performance, several proposals have been made to exploit the semantic information in a transaction to schedule transactions that may not be serializable but still preserve the consistency constraints [30, 57, 71]. Also, there has been some work done to model transactions as operations on abstract data types with built-in operations for recovery, using limited semantic information [81].
Nested transactions have been proposed as a means for implementing reliable distributed computing. Nested transaction model based on dynamic ordering (uses two phased locking) has been suggested by [60]. Reed proposed a static ordering scheme (using timestamps) for implementing atomic actions in a distributed system [68]. The two schemes are pessimistic in the sense that they do not generate non-serializable schedules and if a conflict is detected the operation is delayed or aborted. An optimistic concurrency control for nested distributed transactions has been suggested in [36]. In this scheme, the scheduler optimistically schedules operations, assuming absence of conflict, and verifies the correctness before committing the transactions.

Transactions in Design Databases

The transaction mechanism and models have been extended to model design transactions. Some issues related to CAD databases are addressed in [61]. Other issues involved in providing adequate support for design transactions in a workstation based environment can be found in [26]. Several researchers have attempted to extend the transaction model to support CAD applications [17, 8, 32, 49]. Some of the salient features of these models and their limitations are discussed in detail in Part 4 and Part 5 of this thesis. Version control and change propagation issues are discussed in [20, 12, 40]. The issue of modeling complex objects to support long duration transactions have been addressed in [56, 45, 82].

In the next section a brief outline of the rest of this dissertation along with our contribution is given.
ROAD MAP

The rest of this document is organized as follows. The dissertation is organized into five major parts. These address important issues encountered in design databases. In order to make the individual parts self-contained some discussion of background material relevant to the issues dealt in the other parts will be presented.

Part 1 of the thesis deals with the major issues concerning the design databases [52]. This forms the basis for the rest of the thesis; a background is provided on concurrency control and recovery. We also show why conventional database and transaction models are unsuited for design applications.

In Part 2, we show formally the need for different consistency control mechanisms [54]. In addition to the well known view serializability and conflict serializability, a new notion of correctness - co-serializability is presented. The need for such a correctness criteria in design databases is also elucidated. A formal notion of internal consistency and mutual consistency are presented. We also show how a scheduler can be built based on transaction identifiers (Ids) to obtain schedules that result in internally consistent and mutually consistent objects. In this paper, no semantic information about the data objects are used and the consequence of this approach is that only a subset of possibly correct transaction schedules are accepted. This illustrates the need for a better data model which is dealt in Part 3.

In Part 3, a data model suitable for representing complex objects encountered in design applications is presented [51]. A modeling technique based on entity-
relationship model (E-R) is presented [19]. A complex object is represented as a set of tables. The tables are organized in a hierarchical fashion to stratify the data. Techniques that are necessary for representing design data—abstraction using interface and implementation, multiple versions and instantiation are presented. These concepts are illustrated using examples drawn from VLSI design.

In Part 4, we use the data model and the results obtained earlier to propose a design environment and a transaction modeling scheme [50]. The environment is composed of a central database connected to a set of workstations. Design data are organized into a hierarchy of interfaces and implementation. The hierarchy captures the version hierarchy and instances as well. Analogous to the interface hierarchy is the composite-object hierarchy to capture 'is-a-part-of' relation between complex objects. The database manager associated with the public database is responsible for managing the various hierarchies. Locking modes and conflict table are developed to support operations on the object hierarchies.

The design transactions that are invoked from the workstations run under the auspice of a project transaction which is responsible for maintaining and enforcing its own correctness criteria and can possibly use one of the approaches suggested in [49]. A transaction management scheme for project transaction is presented along with a locking protocol. A type of may-be transactions called hypothetical transactions is introduced. Hypothetical transactions allow designers to experiment with design objects which may often be discarded, without incurring overheads associated with regular project transactions. A scheme for upgrading a hypothetical transaction to
a regular project transaction is also presented.

Part 5 deals with modeling a project transaction into a hierarchy of subtransactions [53]. The model assumes that co-serializability is the correctness criteria in the central database (public database) while conflict-serializability is the correctness criteria for private database associated with each project transaction. The model supports four features essential for design environments:

1. Conventional nested model based on Moss's scheme.
2. Independent commit transactions for avoiding expensive aborts.
3. Conversational interface and communicating transactions for capturing the cooperation between subtransactions.
4. Client-subcontractor transactions for concurrent design activity.

The model allows for cooperation between the subtransaction (designers) and allows the designer to model the transaction system closely to the design environment.

We conclude with a brief summary of the work and suggested areas for future work.
PART I.

ISSUES IN DESIGN DATABASES
ABSTRACT

Transaction management schemes and data models proposed for conventional database applications such as data processing are inadequate for advanced applications encountered in CAD/CAM, VLSI and office automation. The main issues that need to resolved are a new transaction model and control mechanism and a data model. Design transactions cooperate and communicate with other design transactions. This behavior cannot be captured by a transaction model where correctness is based on serializability theory. Due to the multiple versions that coexist in a design database, other correctness criteria - version control, internal and mutual consistency are also required. In addition, issues such as recovery are reviewed in the context of design databases.

Key Words: Design database, data model, serializability, design transactions, version control, consistency.
INTRODUCTION

In recent years computers are playing an ever increasing role in engineering and manufacturing. While programming languages and special hardware have been developed for these applications, classical theories of database systems - originally developed for traditional business and scientific applications - are now being applied for solving problems encountered in engineering applications.

However, databases for engineering and manufacturing applications such as Computer Aided design (CAD), office automation, office information systems, and software development environment, referred to as design databases, have a very different structure and purpose compared to business and administrative databases. The operations or transactions on these databases are also significantly different to warrant a totally new model for these design databases. Researchers and developers of design database systems have dealt with the incompatibility problem by implementing ad-hoc models and mechanisms with limited success [17]. Although such approaches have resulted in several practical systems, in general, they lack a formal notion of correctness [49]. The main reasons for the limited success of design database models are:

1. the wide spectrum of design applications,

2. the nature of design environment, and

3. the complex nature of the design objects.

In this paper, we identify the major issues that are pertinent to design databases. We also review some of the work done in other areas that influence the area of design
databases. The remainder of this document is organized as follows. In Section 2, we
briefly review the background material that is the basis of design databases. In Sec­
tion 3, we review the transaction model and show the limitations of the conventional
model in design applications. In Section 4, issues pertaining to design datamodel
are considered and other issues that influence the design databases are discussed in
Section 5. Section 6 concludes the paper.
ARCHITECTURE

A typical design environment consists of a public database which is also called the design library where design data and artifacts are kept. The data in the public database are shared by all users in the system. The design processes called design transactions are executed on workstations that are connected to the public database through a local area network (Figure 1). The design processes are long running and several design processes may be active concurrently. Since the processes are computation intensive and involve human interaction, in case of failures, recovery issues are important. Complete roll back is expensive and impractical.

The workstations have their own database (private database) that is managed locally by the local database manager. Several designers may work on the design of an object (project); also several design efforts may be in progress at the same time. Different versions of an object may reside at different sites (viz., public and several private databases). Also, several designers may be concurrently designing the same
object, from different workstations, leading to multiple versions of the design object. In addition to the multiple design effort, design alternatives and revisions to existing design objects lead to multiple versions. Design processes or design transactions need to select the appropriate version of the required object from among the set of design objects available.

Therefore, the two major issues in design databases are

(1) Transaction model with suitable correctness criteria, and
(2) Data model to capture the attributes design object.

These are considered in the next two sections.
TRANSACTION MODELS

Transactions are a set of logically related activity on data items, grouped together to form a logical unit of consistency and recovery. In conventional data bases, transactions form the basis of data manipulation activity. They have three important characteristics:

1. Unit of consistency: A transaction takes the database from one consistent state to another.

2. Unit of atomicity: The transactions are atomic. The intermediate steps of a transaction are not observable from outside the transaction. This property is guaranteed even in the presence of concurrent transactions.

3. Persistence: The effect of the committed transactions are permanent. This is in the presence of failures.

Thus, it is desirable to model design activity as transactions, so that data object consistency and reliability can be maintained.

A brief description of classical database model is presented to explain why it is unsuitable for design database.

Classical Transaction Model

A database consists of a set of data entities or objects. Users of the database share access to the objects in the database. A set of assertions or consistency constraints are imposed on the data objects. Example of such assertions on a bank account could be of the form:
Available = Savings \cdot \text{Minimum Balance}

\text{Minimum Balance} > \$100

\text{Transfer} \leq \text{Available}

The state of a database is consistent if all the database entities satisfy the consistency constraints. The objects are modified by operations performed by processes. The set of operations performed by a process is grouped into a sequence called a \textit{transaction}. Examples of transactions in a banking environment are: withdrawal, account transfer, deposit and account balance. Each of these transactions might involve reading and writing of one or more database entities. The transaction model used in tracking databases (classical databases used for data processing applications), \cite{33} are based on the notion that a transaction is an \textit{atomic sequence} read and write operations on the database and hence, is a unit of consistency. That is, a transaction, when executed alone, takes a database from one consistent state to another consistent state. Thus any serial \textit{schedule} of transactions where transactions are executed in a serial order is a correct schedule since it takes the database from a consistent state to another consistent state.

However, to increase throughput and performance, transactions are run concurrently; this may lead to problems as shown below.

Let the database entities be \{A, B\}.

Let the initial values be \(A = 100\) and \(B = 100\).

Let the two transactions be: \(T_1 = \{ t_{11}:A = A + 50; t_{12}:B = B + 50 \} \) and
Let the consistency constraint on the objects be that $A = B$.

If the transactions are executed concurrently in the order $(T_1 \rightarrow T_2)$, then the database moves from \{A=100, B=100\} to \{A=150, B=150\} due to $t_1$ and to \{A=300, B=300\} due to $t_2$. It can be easily seen that the consistency constraint is maintained by both the transactions. However, if both the transactions are executed concurrently in the order $(t_11 \rightarrow t_21 \rightarrow t_12 \rightarrow t_22)$, the sequence of the database states are: \{A=150, B=100\}, \{A=150, B=200\}, \{A=300, B=200\} and \{A=300, B=250\}. Evidently, the new schedule of operations now do not maintain the consistency constraints. Thus, access to shared data has to be controlled.

Several concurrency control mechanisms have been proposed [27, 64]. These mechanisms also called scheduler, in general are based on Serializability theory. Since transactions in the absence of concurrency maintain consistency, serial execution of transactions or serial schedules are correct schedules and hence acceptable. Serializability theory requires that only those execution sequences or schedules equivalent to a serial schedule be accepted by the concurrency control mechanism.

Transactions send their requests to the scheduler. The scheduler may allow the operation immediately, delay it in case of a conflict or abort the transaction making the request. Schedulers based on serializability theory use transaction and operation syntax to schedule operations. For example, in two-phased locking scheme [27] where transactions have to obtain a lock on an entity before performing an action on the entity, the transactions have to be two phased — a growing phase where transactions
can acquire locks, followed by a shrinking phase where transactions release their locks without acquiring any more locks.

Nested Transactions

Nested transaction model, introduced by Moss [60] allowed the structuring of transactions to improve concurrency and reliability in a distributed environment. In this model a transaction is represented by a tree like hierarchy of subtransactions with the operations forming the leaves. Moss’s model requires serializability at every level of the tree and uses locks. Reed proposed a scheme which uses multiple versions and timestamps to provide concurrency control [68]. In Reed’s scheme objects are represented by a history of object versions. Access to the objects (read and write) are allowed based on the operation type and the timestamp values associated with the transactions and the data objects.

However, models based on serializability theory have been found to be too restrictive [35]. In addition, determination of serializability was shown to be NP-complete [64]. Several models based on transaction semantics have been proposed, which improved on the class of schedules accepted by serial schedulers. Garcia-molina[30] proposed a scheme were serializability can be relaxed leading to non-serializable schedules that preserved consistency. Another paradigm for concurrency control in databases exploit semantic information in transactions and using consistency constraints on the data entities, schedule operations that may not be serializable, but still maintain consistency. Lynch [57] proposed a nested transaction model based on Multi-level
Atomicity where transactions are partitioned into classes based on break-points, and an hierarchy of these classes are established. Users within a class were allowed to have a higher degree of interleaving compared to transactions outside the class. While such models allow a richer class of schedules than those based on serializability theory, they have two serious drawbacks. First, they are not modular: a modification of one component in the system (e.g., addition of a transaction class) necessitated the restructuring of the entire system. Second, they require the users to provide substantial information to the system such as breakpoint specification. In addition, in such systems, the burden of proving the correctness of the resulting schedules rests on the programmer/user.

In the models discussed above, if an operation of a transaction is found to leave the database in an inconsistent state, the scheduler aborts the transaction undoing the effects of all the operations performed by the aborted transaction. This is called rollback or backward recovery.

These models, developed for business applications are unsuitable for design databases: they are either too restrictive or they do not consider the requirements of the design databases as explained in section on correctness criteria.
DATA MODELS

For an efficient implementation of a CAD/CAM database, a data model capable of representing the design object in a convenient form, suitable for easy manipulation is imperative. A design object is a set of attribute values that is treated as a single entity at some level of abstraction in a design environment. The attributes themselves may be atomic (elementary data type such as integer, character), or a set of other attribute values. For example, a VLSI circuit can be treated as a design object, which is in turn composed of other basic circuits. A complex design object may be designed in stages using simpler objects. A designer may design the basic building blocks which can then be used in the design of more complex objects. This approach, called bottom-up design, is natural and is prevalent in design environments. Thus the representation of design objects have to be stratified- allowing for access and manipulation of data at different levels of abstraction.

Inadequacy of Conventional Data Models

Design objects cannot be represented by a simple scheme used in conventional databases such as the relational model with tuples of atomic attributes (First Normal form or 1NF). Another problem associated with the relational model is the lack of a suitable mechanism to capture hierarchically structured data that are often encountered in advanced database applications such as CAD/CAM.

The inability of the relational model, in its current form, to capture complex objects is one of the limitations of the model. Several extensions have been proposed to the relational model [56, 22]. Researchers have also attempted to model complex
objects using E-R modeling technique [Chen] in which the complex object can be decomposed into a hierarchy of relations [10]. Other attempts at modeling complex objects include non-first normal forms (NF²) [2] and object-oriented approaches. These approaches tend to reduce the semantic gap between the data model and the world that is being modeled. In recent years object-oriented programming has been receiving considerable attention. Several object-oriented databases have either been built or under development [Orion, Iris, Gemstone]. Object-oriented approach appears to offer some advantages over other models [58]. Although there is considerable experimental work in the area of object-oriented databases and the field is rapidly evolving, two important issues remain: (1) the lack of a common data model, and (2) the lack of formal foundation unlike the relational model [4].

Issues in Data Modeling

The data involved in processes under our consideration include the following forms: textual data in the forms of manuals, specifications, schedules, project and process plans, numeric data from analytic experiments and calculations, formulas and part geometries, and graphic data in the form of drawings, sketches and technical documentation. These data have to be mapped to entities or elements that can be maintained in a database. Such mapping of objects to elements of a conventional database model (relational, network or hierarchical) is difficult at best and in general impossible. The issues that need to be considered in the design/selection of a suitable data model are listed below.
1. Complex Objects: The design artifacts usually are composed of a collection of heterogeneous records which together represent the design artifact - a complex object. The set of records cannot be conveniently represented by a small set of homogeneous records or tables of tuples as in the case of conventional database entities.

2. Composite Objects: Composite objects are complex design objects formed by integrating simpler objects that are defined independently, resulting in a composition hierarchy. The data model should facilitate the creation and maintenance of such a composition hierarchy. Note that all composite objects are complex objects as well.

3. Sharability of Objects: For practical considerations such as economy, efficiency of design effort and storage, composite objects are designed using components that are defined and available in a design database accessible for designers. The components in the library are shared by all designers. The model should facilitate such sharing. In addition, a composite object might use several instances of an object in its design. Therefore, during representation, such redundancy needs to be avoided.

4. Multiple Versions: Typical design processes involve creation and testing of multiple versions - both as design alternatives and as revisions to existing design. Thus several versions may exist for each ‘object type’. The data model should facilitate the creation and maintenance of such a composition hierarchy.

5. Abstraction Hierarchy: In a system where there are users at different levels with different views of the data, a hierarchy of data abstraction is essential for
the users to access and manipulate the objects. In addition to the ease of use in handling and managing the complex data, an abstraction hierarchy allows for the stability of the system. Each abstraction level acts as a firewall for changes in the data as they evolve - which is often the case in a design database where all changes at lower level may not propagate up in the data hierarchy.

6. Operations on Objects: The operations on the design objects also pose some problems. The complex objects may have to be retrieved and manipulated as a whole or a part of the object may be modified. In both cases, the data model should facilitate the operations efficiently.

7. Multiple Representation: An object may have different interpretations in different 'views'; for example a circuit diagram could be viewed as a set of connectivity information or a graphic layout, depending on the environment.

8. Restructuring /Schema Change: The design objects are usually dynamic; apart from the changes in their attribute values, they undergo frequent structural changes as well. Thus, this may necessitate restructuring of the data organization. In conventional databases, this is similar to changing the schema definitions. In Object-oriented systems, this is equivalent to changing the class object. The model should not only facilitate online changes to the object structure, but also propagate the changes to other objects that might be influenced by the change. This is called schema evolution and change propagation. These issues in object-oriented systems are discussed in [9].
CORRECTNESS CRITERIA

Serializability has been found to be too restrictive. However, some control over the transactions have to be maintained to preserve object consistency. In particular, due to the presence of multiple versions of design objects, all of which are correct at any given time, several other issues arise. These are discussed below.

We assume that all transactions when operating independently read version consistent objects and create version consistent objects. While view-serializability has been the approach taken by schemes based on object check-in/checkout protocols, we maintain that co-serializability (called $\tau^*$-serializability in [78]) is sufficient for maintaining design database consistency.

View Serializability and Co-serializability

View-serializability is the correctness criteria enforced by conventional schedulers in tracking databases. Let $T = \{t_1 \ldots t_n\}$ be the set of committed transactions in a schedule $S$. The schedule $S$ is view-serializable if and only if there exists a serial schedule $S'$ composed of transactions on $T$ such that each transaction $t_i \in T$ reads and writes the same values in both $S$ and $S'$.

While view-serializability is necessary for tracking databases (used for maintaining a track of data values), as argued earlier, it is too restrictive for design applications. Design applications generally involve generation of several alternate designs, possibly simultaneously, until the final design is obtained. Thus the design transac-
tions can be represented by a history graph HG as shown in Figure 2, where an arc \((t_i, t_j)\) denotes the dependency relation between the transactions.

In the history graph HG, \(t_0\) is the initial transaction which initializes the database and \(t_{12}\) is the final transaction which chooses to use the values generated by \(t_5\). The design objects generated by transactions in \(\{t_6, t_9, t_{10}, t_{11}\}\) are not used and can now be discarded. To ensure correctness of transaction \(t_{12}\), it is sufficient to maintain serializability only for the set of transactions that influences it; that is for \(T' = \{t_0, t_1, t_2, t_4, t_5, t_{12}\}\).

Let \(P\) be a subset of \(T = \{t_0, \ldots, t_n\}\). A schedule \(S\) of transactions in \(T\) is \(P\)-serializable if the values read (and hence written) by the transactions in \(P\) are the same as those in a serial schedule \(S'\) involving the same set of transactions \(T\) [78].

Definition: A schedule \(S\) of transactions in \(T = \{t_0, \ldots, t_n\}\) is co-serializable if
it is $t_i$-serializable for all $t_i \in T$.

Co-serializability can be achieved by a two-step protocol where all reads of a transaction are performed before writes. Thus by requiring the transactions to write objects into the public database as the last step before ‘commit’ **.

Version Control

The recent surge of interest in computer aided design (CAD) and other related areas has resulted in several proposals that address the issue of version control [39, 12, 79, 20]. In most of the suggested models, distinction is made between two instances of an object: the released version residing in the public database, and the working version that is being derived by a transaction. A comprehensive discussion of the issues related to version control is presented in [20]. However, other models consider only those issues restricted to change notification or propagation while in [79] change propagation is not considered. Design processes involve creation and manipulation of several versions of design objects. Therefore version control is an important function that has to be supported by a design database. There are several issues pertaining to the problem of version control. The three most important problems are listed below:

1. internal consistency,

2. mutual consistency, and

3. referencing and dereferencing (binding of object to the name).

Since these issues are paramount in design databases, we will now describe them in greater detail.
Internal Consistency

In design databases, the design objects are maintained by a central database. Transactions (design applications) access these objects and use them to 'derive' other complex objects. Since complex design efforts are often iterative, other transactions may read data objects and redefine (refine) them creating alternate versions. Also, transactions using different design principles/techniques may be used concurrently to develop alternate versions of which only one may be used in the final design. Therefore, one can envision a history of versions associated with each object. Since several versions exist for each object, during derivation of a complex object, only consistent versions of objects need to be used. Informally, internal consistency deals with restricting the set of objects read or used during the derivation of an object.

Consider for example, the process of compiling a book consisting of several chapters, each of which is written by a different author. Each section of the book — the chapters, the contents, the bibliography etc., represent derived objects. The compilation of the bibliography and the content section are derived by transactions that use all the chapters in the book to obtain the necessary information. If one of the chapters, say Chapter 2, is modified by its author, by addition or deletion of some sections, resulting in a new Chapter 2', the other sections of the book, (in particular, the contents and the bibliography) become incompatible with the new Chapter 2'. Unless the contents and the bibliography sections are rederived to reflect the changes in Chapter 2, the compilation of the book should not include the new version of Chapter 2, as it is not compatible with the rest of the book. Note that under conventional schemes the latest values would have been used during the derivation of the
book including the new Chapter 2' leading to inconsistent final object.

Version control mechanisms enforce such constraints generally encountered in design database applications. None of the existing schemes have an automatic version control; they require explicit control information from the users. This requires the user to keep track of the different versions of all the data objects that are being used, placing considerable burden on the user.

**Mutual Consistency**

The issue of mutual consistency deals with the generation of a set of internally consistent objects, and hence can be considered as the dual to internal consistency problem. This issue has been discussed in literature as change propagation or change notification. Most of the literature available treat only this aspect of version control.

In design databases, a design object is composed of (derived from) other simple objects which are again composed of other simpler objects leading to a hierarchy of design objects called *configuration hierarchy*. We assume that the hierarchy can be represented by a directed acyclic graph (dag). We restrict out transactions to those based on a dag-configuration. The configuration hierarchy of a book is shown in Figure 3.

In a hierarchy, if one of the lower level object is changed, then the higher level objects need to be made aware of the change. Consider the example of the derivation of a book. Let a transaction read all the chapters (version consistent) of the book,
Figure 3: Configuration Hierarchy of a Book

revise them if needed, and then generate the table of contents (TOC) and the index (IND), and then output these along with the revised chapters. The output of this transaction may then be used to print the book. The transaction may be modeled as a set of cooperating (concurrent) processes or subtransactions each attempting to revise a chapter and the result of these subtransactions may then be used to derive TOC and IND. Assume that after a set of revisions, TOC and BIB are derived; now if one of the chapters say Chapter 1 was found to be in error (which is very likely) and was revised by one of the subtransactions, then the new version (Chapter 1') is not mutually consistent with TOC and IND. However, note that the other chapters and the old version of Chapter 1 are (mutually) consistent with TOC and IND. Also, while each of the objects (the chapters, TOC and IND) are internally consistent they are not mutually consistent! Now to maintain mutual consistency the system can do one of the following:

- Disallow the creation of a new version of Chapter 1 once TOC and IND have
been derived.

• Inform the designer/process in charge of TOC and IND of the change (notification).

• Automatically create a new version of TOC and IND (propagation).

• Destroy TOC and IND.

The exact choice of action depends on the context. Other issue that needs to be addressed is the ‘depth’ of notification or propagation. That is, how deep into the hierarchy should the effect of the change be propagated.

**Referencing Objects**

Since several versions of an object co-exist in the database, it must be possible for a designer/transaction to reference a particular object version. Also, the users should not be bogged down with the details of naming conventions etc., — the naming scheme should support explicit and implicit naming. When necessary one could selectively access any object version by explicitly naming it; however one could avoid using a specific name.version by using implicit naming scheme by referring to the object by its generic name.
OTHER CONSIDERATIONS

Cooperating Transactions

Large design efforts are partitioned into a number of projects which may all need to share some data objects. That is, in database notation, transactions may be decomposed into subtransactions which may then access the common objects. Thus, a design project is a result of a joint effort of several subtransactions, each aware of the other, cooperating by exchanging intermediate object values. However, conventional models, where a transaction is modeled as a sequence of atomic operations, do not promote such a cooperative design environment.

Several of models that have been proposed earlier suffer from serious limitations. For example, [39, 56] model design transactions as a sequence of conventional (short duration) transactions, where a failure of a short duration transaction returns both (public and private) databases to the consistent state that existed before the failure. Furthermore, they do not handle the notion of multiple object versions. In [47], the notion of database consistency that is satisfied is not properly established. The models do not support or promote the concurrency and cooperation between subtransactions necessary in a design environment. Also, the assumption that a design transaction is a sequence of short duration transactions is not necessarily valid as it precludes the notion of cooperating transactions.

Long Transactions

While transactions in data processing applications are short, typically running for a few seconds or less, transactions in design databases are long-running. The
‘design transactions’ may execute for several hours or even days, with designers interact- ing with each other, iteratively developing new designs. Hence, methods like two-phase locking that require transactions to hold locks during most of their executions, while suitable for conventional databases are unsuitable for design databases, as such schemes will lead to poor throughput. In addition, probability of deadlock is proportional to the fourth power of transaction duration [34]. Thus, the rate of occurrence of deadlocks under conventional schemes, in design databases would be unacceptable.

**Failure and Recovery**

All practical systems, as we know, are prone to failures. Failure may be attributed to (but not limited to) one of the following reasons.

1. Hardware failure such as communication link failure, site crash, and disk crash.

2. Software failure due to programming errors.

3. Deadlocks leading to termination of one or more processes.

4. User errors such as typos, wrong inputs.

Failures lead to abortion of one or more transactions in the system. In case of failure, the system should be able to rollback the system state to maintain the atomicity of transactions (failure atomicity). As pointed out earlier, transactions in design applications are generally long-running and computation intensive. Thus, aborting transactions are expensive since this not only involves undoing all the changes made to the data objects, but also results in wasting of computations that have been performed so far. Ideally, schedulers in design environments should avoid transaction
abortions. However, in the event of a failure, in design applications, the system should rollback to an intermediate state to minimize the recovery cost.

Modularity Requirements

Typically, in design environments new transaction types are routinely introduced. Thus, the requirement that all transaction classes and concurrency control mechanisms be redefined for every such modification or addition to the transaction class is unacceptable. Therefore, the schedulers for design databases must be modular.

Schedulers based on semantic information (e.g., \([30, 57]\)) are static in that the type of transactions that may be scheduled in them are fixed; any addition to the transaction class or modifications to the existing transactions may require extensive modifications to the entire transaction class and concurrency control mechanism.

Conclusion

In this paper, we proposed a model of a design database suitable for advanced design applications such as those encountered in CAD/CAM, software engineering, and office automation. We also discussed the limitations of the conventional transaction and data models designed for data-processing applications. The issues that need to be resolved in a successful design database are also discussed. A pre-requisite for a successful design/implementation of any system is the understanding of all the underlying issues. It is hoped that this paper would serve to that end.
PART II.

TRANSACTION MODELS FOR DESIGN ENVIRONMENTS
ABSTRACT

Design databases used in the design of engineering artifacts have characteristics quite different from conventional databases. While conventional databases have at any given time a single correct value for each of its objects, in design databases, due to the presence of design alternates or versions, several correct versions of design objects are available. However, during a design process, when several design objects are involved, the different versions of each object that is chosen for the design has to be controlled such that they are compatible or version consistent. The paper formalizes the notion of version consistency and proposes two transaction models for design processes, where the notion of version consistency itself is used as a basis for scheduling the design transactions. A mechanism for enforcing version control is also presented.
INTRODUCTION

Design databases – databases used in applications such as CAD, VLSI design, and software development for storing design data – are, in recent years, receiving considerable attention [56, 47, 26]. Design databases differ from conventional databases in several ways: the type of data objects stored, the nature of design processes, and the design environment itself [61].

The classical model of transactions and correctness criteria as suggested by Gray and Eswaran et al. is widely used in conventional database applications such as data-processing and banking [34, 27]. Database systems designed primarily for conventional database applications are not suited for design database. The classical model is based on serializability; only those schedules that are view equivalent to a serial schedule are acceptable and hence scheduled [64]. Thus, each data object at any time in its history has a unique value associated with it. In design environments, several design teams may concurrently be involved in the design of an object, leading to multiple versions of an object, and each of these versions might be “independently correct” and can be used by other design processes to create versions of other objects. However, a set of objects may not be compatible with each other as they may have been derived using different (and therefore incompatible) versions of the same objects. Thus, in order to maintain version consistency, design processes that use design objects to create other design objects need to select “versions that are compatible.” Schemes that have been proposed earlier (see for example, [20, 12]) do not discuss this aspect of version compatibility; they require the users to explicitly name the required object versions and this necessitates the users to keep track of the
different versions of the objects that are compatible.

In this paper, we establish a formal notion of version compatibility which is used to abstract the notion of versions from designers. Vidyasankar proposed a new notion of correctness for design database transactions called \( r^*_s \)-serializability in [78], where each transaction in a schedule has the same view of the database as it would have in some serial schedule containing that transaction. While testing for \( r^*_s \)-serializability in arbitrary schedules is NP-complete, schedules in which all transactions maintain version consistency can be tested in polynomial time. To this end, we propose a mechanism for maintaining version consistency. We also show that given a set of design processes that can derive a set of design objects, no on-line scheduler can, in polynomial time, schedule a set of processes that can generate a set of version consistent objects. We present two models of design transactions in which the correct schedules are a subset of \( r^*_s \)-serializable schedules, and show that schedulers based only on version consistency are sufficient for scheduling correct schedules.

The remainder of this paper is organized as follows. In the next Section, a brief discussion of the design environment is presented. In Section 3, we discuss the uniform transaction model. The nonuniform model is presented in Section 4. In Section 5, we propose a mechanism for enforcing version control. Section 6 concludes the paper.
DESIGN ENVIRONMENT

In this section, we briefly discuss the design environment. Some of the characteristics of design processes are discussed below.

• **Long Duration Processes.** Design processes, unlike transactions in conventional databases, are long-running; they may execute for several hours or even days, and may extend over several ‘login sessions’.

• **Complex Design Objects.** Design objects – design entities that are manipulated by design processes– are complex. They can be unstructured and large, making it difficult to represent them by single flat tuples.

• **Object Hierarchy.** Design objects often are derived from other objects, resulting in an object hierarchy of data objects.

• **Multiple Versions.** Design processes are often iterative with new versions being created during every cycle. In addition, multiple versions may be needed as design alternates.

• **Cooperative Effort.** Design processes are typically a cooperative effort of a team of people who share data and interact with each other. The team itself can be subdivided into several smaller groups; each group may view the design objects at different levels of abstraction (abstraction hierarchy). It is necessary to coordinate the efforts of all members of the team.

Consider for example, the VLSI design process. It is representative of a fairly complex design process – the design effort is complex, involving efforts of humans
and CAD tools, and requiring generation and manipulation of large amounts of data. A set of tools called silicon compilers provide CAD support for the design effort [70]. The structure of a typical silicon compiler is shown in Figure 1. While it is the goal of the silicon compilers to totally automate the process of VLSI design, design capability still lags behind production capability and human interaction and supervision are still required. From the initial input which is generally a file containing hardware description in a hardware description language, the system goes through several synthesis and analysis steps to arrive at the final result – the chip mask and other control data necessary for making the integrated circuit. The synthesis tools help perform routine tasks such as routing, routing optimization, and programmable logic array (PLA) generation; the analysis tools are used to detect design rule violations and logical failures.

We can now model the design activity in such an environment. Design objects can be represented by complex objects [7]. We do not impose any other constraints on the objects other than the requirement that all complex objects involved in the design should be representable by the chosen scheme. The objects that are used to represent the initial inputs (e.g., specification, design rules, library) are called base objects and other objects that are derived by design processes are called derived objects. Design objects are shown as circles in Figure 1. The design processes are called transactions in the design database and are denoted by rectangles in Figure 1. Design transactions may span several ‘sessions’ or invocations of a program. For example, the process of modifying the high-level specification may involve a session with the interactive editor, requiring invocation of several edit and file manipulation commands.
Typically, in a design environment, all objects generated during the final stage may not be selected. Since there may be several alternatives to obtain a final objective, several designers may develop independent "versions" of each object. For example, a team of designers may, using the same high-level specification, develop several versions of an integrated circuit (IC), independently. A new pad-placement and pin configuration may be obtained for the same circuit by using different connection specifications (refer to Figure 1). Similarly, when different routing specifications are used, alternate pin/pad placements are generated, resulting in multiple versions of the IC mask, for the same high-level specification. When the different versions of the IC are tested, (using possibly different versions of test inputs), a version of "result" is produced for each combination of IC version and test-input. When analysis is done on the mask or the pad-placement result, the analysis process must be able to identify the versions of connection specification or the routing specification, with the corresponding final mask or the pad-placement. That is, only compatible versions of objects need to be used by the test/analysis process. In subsequent sections we identify other issues that motivate the need for version consistency.

In a design environment, feedback is often used to modify the objects; a derived object may thus influence the base objects - leading to redefinition of the specification. The feedback loop leads to cycles in the derivation scheme. In this paper, we assume that the feedback loop is outside our design transaction model, and consequently the derivation schemes are assumed to be acyclic graphs.
Based on the model of objects used by the transactions and the relationship between objects, two schemes are distinguished in this paper: (1) the uniform model, and (2) the nonuniform model. These models are discussed in the subsequent sections.
UNIFORM TRANSACTION MODEL

The composition hierarchy of objects in a design scheme can be represented by an acyclic directed graph \((O, E)\), where \(O = \{O^0, \ldots, O^n\}\) is the set of objects and \(E\) is the set of edges, such that there is an edge \((u, v)\) in \(E\) if \(u\) is used during the derivation of object \(v\). The set of initial inputs or the base objects can be represented by a complex object \(O^0\) and the final design objective by a complex object \(O^n\). The objects in \(\{O^1, \ldots, O^n\}\) are the derived objects. The design scheme is \(n\)-staged if \(|O| = n + 1\. The design scheme is said to be \textit{non-redundant}\) if there is a path from every object \(O^i\) to the final object \(O^n\).

Let \(DG\) be an instance of a derivation scheme. Then we define the following:

**Definition:** \(\text{Dep}(DG)\) is a relation over \(O \times O\) such that if there is a directed path from \(O^i\) to \(O^j\) in \(DG\) then \(i, j\) is in \(\text{Dep}(DG)\). If \(i, j \in \text{Dep}(DG)\) then object \(O^j\) is a dependent of \(O^i\).

Every object \(O^i, i \neq 0,\) is a dependent of the initial object \(O^0\). In a nonredundant derivation graph, the final object \(O^n\) is dependent on every object \(O^i, i \in \{0, \ldots, n-1\}\).

**Definition:** \(\text{Indep}(DG)\) is a relation over \(O \times O\) such that \(i, j \in \text{Indep}(DG)\) if \(i, j \in \text{Dep}(DG)\) and \(j, i \in \text{Dep}(DG)\).

Given an instance of a derivation scheme \(DG = (O, E)\), our intent is to derive the final object \(O^n\), starting from the initial object represented by \(O^0\). We shall assume that each intermediate object \(O^i\) is derived by a single transaction \(T^i\). Let \(DG\) be an \(n\)-staged derivation graph and \(S\) be a possibly interleaved schedule of transactions in \(T = \{T^1, \ldots, T^n\}\) for the derivation graph \(DG\).
**Definition:** Schedule S is said to be *synchronous* if it is serializable in some order consistent with Dep(DG). That is, if \(<i,j> \in \text{Dep}(DG)\) then \(T^i\) precedes \(T^j\) in S.

For example, let \(T = \{T^1, T^2, T^3, T^4\}\) be the set of design transactions for DG' and \(\text{Dep}(DG') = \{<1,4>, <2,4>, <3,4>, <1,2>\}\). If a schedule of T is serializable in the order (1;2;3;4) or (1;3;2;4) or (3;1;2;4), then it is synchronous.

**Definition:** Let DG = \((O, E)\) be a non-redundant design scheme where \(O = \{O^1, \ldots, O^n\}\) is the set of design objects, \(T = \{T^1, \ldots, T^n\}\) is the set of transactions, and \(\leq\) is a partial order on the transactions in T. The transaction model \(\cdot \cdot, \leq, O \cdot \cdot\) is *uniform* if:

1. every \(T^i \in T\) reads at least one object \(O^j \in O\), and
2. every \(T^i\) writes exactly one object \(O^j \in O\).

It follows from the definition that in a uniform model, no two transactions write on the same object.

While an object may be found to be semantically correct, it is *acceptable* only if it can be derived by a synchronous schedule. This requirement is similar to that imposed in conventional databases where view-serializability is the correctness criterion although a non-serializable schedule might produce a consistent state. While this may be restrictive, in the absence of other semantic information about the objects involved and their correctness criteria, such an assumption is reasonable. Consequently, we have the following:
Preposition 1: A synchronous schedule derives an acceptable design object at every stage.

Multiple Versions

Typically, in a design environment while only a single instance of a final object may be needed, multiple versions of the design objects may be derived due to the following reasons:

1. Several versions may be released as final design.

2. Alternate design of each object may be developed during design phases.

3. Alternate design may be used for refinement of design by 'evolution'.

Thus each design transaction $T^i$ can now be replaced by a set of transactions \{${T_1^i, .., T_{mi}^i}$\} representing alternate design for the design stage for $i = 1...n$. Multiple versions of the design object can now be obtained by scheduling synchronous schedules $S_1, S_2, ..., S_k$ sequentially where each schedule $S_i$ consists of a set of transactions = \{${T_{i1}^1, .., T_{in}^i}$\} as shown in case(a) of Figure 2. Another alternative is to require a transaction $T_{jk}^i \in T^i$ to derive version $O_{jk}^i$ of object type $O^i$. Thus no two transactions write to the same object version and therefore we could run the different schedules $S_1, ..., S_k$ concurrently. Now under this scheme, each object $O^i$ produced by a transaction in stage $i$ is replaced by a set of versions of the object: \{${O_{jk}^i, T_{jk}^i \in T_{ij}}$\} as shown in case(b) of Figure 2.
Let \( T^i = \{ T^i_1, \ldots, T^i_{m_i} \} \) be the set of alternate design transactions available for stage \( i \). Let \( T = \bigcup_{i=1}^{n} T_i \). We now define the following.

**Definition:** \( Sel(T) = T^1 \times T^2 \times \cdots \times T^n \).

\( Sel(T) \) is a set of transaction sequences formed by selecting a transaction from the set \( T_i \) for each stage \( i \). It follows from preposition 1, that any synchronous schedule \( s \in Sel(T) \) produces an acceptable final object. In a uniform transaction model, every version of an object of type \( O^i \) has the same set of attributes (structure) but possibly different attribute values.

**Coherent Schedules**

Let \( S \) be an interleaved schedule of \( T \). The schedule \( S \) is said to be *coherent* if every object \( O^n_{jk} \) is acceptable. Since an object \( O^n_{jk} \) is acceptable if it is derived by a synchronous schedule \( s_k \in Sel(T) \), we require each transaction in \( S \) to read (and hence write) the same set of object values as it would have in a synchronous schedule containing the same set of transactions. In a design environment several groups may be attempting to design the same set of objects concurrently resulting in a schedule \( S \). We would require each of the final designs produced to be acceptable; we would prefer \( S \) to be coherent. One approach to this problem could be to generate members of \( Sel(T) \) and schedule each of them in an independent environment. Another approach could be to rename the read-sets and write-sets of each transaction such that they could be executed without interference in the same environment. However these schemes may not be acceptable due to the following reasons:

1. The set of schedules that may need to be scheduled may not be known a priori.
2. Identical version of the object may be rederived independently in different environments, resulting in wasted computation.

3. It may be necessary to recompile each transaction belonging to different schedules with different names.

4. It may be necessary to maintain the set of related objects in the same place to facilitate configuration management [5].

For reasons of efficiency, we would maintain all versions of the objects in a shared database for reusing them. Revision control systems such as Unix RCS [76] can provide adequate support for maintaining the different versions. In addition, we would abstract out the notion of version names (the suffixes) associated with the object. This will obviate the need for recompiling the transactions for each schedule and would facilitate a consistent naming scheme for objects manipulated by the transactions. The version control could be done by the system such that each final object produced by a schedule T is acceptable.

Note that in the case of software development, our notion of consistency is not merely to maintain the object version consistent with the source code - a facility provided by programs such as make [28], but also to maintain several such versions of the final object, in the presence of complex object dependencies. This aspect is discussed further in section 5. In the following section we introduce the notion of correctness that is necessary for determining the appropriate versions of each object such that a schedule S may be coherent.
Object Consistency

In the models we consider we assume that all transactions read a nonempty set of data objects and write a nonempty set of data objects. The read-set of a transaction \( T^i \), denoted \( RS(T^i) \), is the set of objects that are read by the transaction \( T^i \); the write-set of transaction \( T^i \), denoted \( WS(T^i) \), is the set of objects written by the transaction \( (O^i) \). Without loss of generality, we assume that \( RS(T^i) \subseteq \{O^0, O^1, \ldots, O^{i-1}\} \). Note that such an assumption is valid because the derivation schemes we consider are acyclic. We will now recursively define internal consistency of an object and mutual consistency of a pair of objects. Informally, an object is internally consistent if it is in an acceptable state, and has reached that state through a set of acceptable steps, whereas any two objects that are internally consistent are mutually consistent if they are 'compatible'. Note that when there are design alternates for each design object, all internally consistent objects need not be mutually consistent with each other. As an example, consider the implementation of a VLSI circuit. Many different circuits may exist for the same functionality. Thus, the routing information and connectivity details of a design may be mutually consistent with each other if they correspond to the same design alternative, but may not be mutually consistent with those obtained by an alternate (but still correct) design. The notions of consistency are formally defined below.

**Definition** If all objects \( \in RS(T^i_j) \) are pairwise mutually consistent and internally consistent then \( O^i_j \in WS(T^i_j) \) is internally consistent.

Mutual consistency is defined recursively as follows:

1. The initial object \( O^0 \) is mutually consistent with every object \( O^i_j \).
2. Any two versions of an object are not mutually consistent: \( O_j^i \) is not mutually consistent with \( O_k^i \) if \( k \neq j \).

3. If \( \forall v, \forall u, u,v \in RS(T_j^i) \), \( v \) and \( u \) are internally consistent and mutually consistent then \( WS(T_j^i) \), that is \( O_j^i \), is internally consistent and mutually consistent with \( u \) and \( v \).

4. If \( (i,j) \in \text{Indep}(DG) \), then \( O_k^i \) is mutually consistent with \( O_j^i \) if every object in \( RS(T_j^i) \) is mutually consistent with every object in \( RS(T_k^i) \).

**Theorem 1** Let \( S \) be a schedule of \( T \). \( S \) is coherent iff every transaction in \( S \) reads objects that are internally consistent and pairwise mutually consistent.

**Proof:**

Claim 1: \( S \) is coherent \( \implies \) \( T_j^i \) reads mutually consistent objects.

Proof of claim 1: \( S \) is coherent \( \implies \) \( O_j^n \) is acceptable.

\( O_j^n \) is acceptable \( \implies \) \( O_j^n \) is produced by a synchronous schedule of transactions in \( \{ T_{j_1}^1, T_{j_2}^2, \ldots, T_{j_m}^m, T_{j_1}^n \} \). We have

\[
\begin{align*}
\text{RS}(T_{j_1}^1) & = O^0 \\
\text{RS}(T_{j_2}^2) & \subseteq O_{j_1}^1 \cup O^0 \\
& \vdots \\
\text{RS}(T_{j_1}^n) & \subseteq O_{j_m}^m \cup \text{RS}(T_{j_m}^m).
\end{align*}
\]

From the definition of mutual consistency, it follows that \( T_{j_1}^1 \) reads mutually consistent object \( O^0 \). The read-set of \( T_{j_2}^2 \) is \( \{ O^0, O_{j_1}^1 \} \) and from the definition it follows that they are mutually consistent resulting in \( O_{j_2}^2 \). Similarly it can be shown that each of the transactions in the schedule reads mutually consistent objects.
Claim 2: Every \( T_j^i \in S \) reads mutually consistent object \( \Rightarrow S \) is coherent.

Proof of claim 2 is by induction on the number of design stages \( n \).

**Basis** \( n = 1 \). Trivial: Every transaction \( T_j^1 \in S \) reads mutually consistent object(s) \( O_j^0 \). Every object \( O_j^1 \) is produced by a synchronous schedule \( \cdot T_j^1 \cdot \) and hence is acceptable. Thus \( S \) is coherent.

**Induction** Let it be true for all design stages up to \( m \) stages. Let \( S \) be a schedule with \( n = m + 1 \) stages. Let \( S' \) be a schedule same as \( S \) with transactions in \( T^n \) removed. Consider object \( O_j^m \) produced by a schedule \( S' \). Since every transaction in \( S' \) reads mutually consistent objects and \( S' \) has \( m \) stages, \( O_j^m \) is produced by a synchronous schedule. Without loss of generality, let the schedule be \( s_j \) of transactions in \( \{ T_1^j, \ldots, T_m^j \} \). Let \( T_k^n \) read object \( O_j^m \) to produce \( O_j^n \). We need to show that object \( O_j^n \) is produced by a synchronous schedule of transactions in \( \{ T_1^j, \ldots, T_m^j, T_k^n \} \).

**Case 1:** \( RS(T_j^n) = O_j^0 \cup O_j^m \).

Then schedule \( \cdot s_j ; T_j^n \cdot \) is also synchronous.

**Case 2:** \( RS(T_j^n) = O_j^0 \cup O_j^m \cup O_k^i \)

Since \( T_j^n \) reads mutually consistent objects, by definition of mutual consistency, if \( O_{ji}^i \in s_j \) then \( k = ji \) and the schedule \( s_j ; T_j^n \) is synchronous. However, if \( T_k^i \neq s_j \) and if the read-set of \( T_j^n \) consists of mutually consistent objects, then it follows that \( s_j \) such that \( T_j^i ; T_j^n \). Therefore the schedule \( \cdot s_j ; T_j^i ; T_j^n \cdot \) is a synchronous schedule. \( \blacksquare \)
Our notion of coherent schedule is the same as $\tau*$-serializability proposed by Vidyasankar in [78] restricted to the uniform transaction model. Furthermore, it was shown in [78] that determination of $\tau*$-serializability is NP-complete. Thus in the context of uniform transaction model, through Theorem 1 we show that maintaining version consistency is tantamount to obtaining $\tau*$-serializability. A mechanism for version control is presented in section 5.
NONUNIFORM TRANSACTION MODEL

In the uniform transaction model, it was assumed that each transaction $T_j$ in a schedule generates only one version of a complex object $O^i_{j,k}$. While it is possible to represent a set of objects by a complex object, it may be too restrictive for some applications. Also, under uniform transaction model, it was assumed that each object type $O^i$ is written by transactions belonging to $T^i$. A nonuniform transaction model consists of a set of transactions $T = \{T^1, \ldots, T^n\}$, and a partial order $\preceq$ on $T$. The nonuniform transaction model differs from the uniform transaction model in the following ways:

- The write-set of each transaction $t \in T$ may contain multiple objects.
- The final design objective may be a set of objects $\subseteq O^D$.
- The write-set of any two transactions in $T$ may intersect.

Let $DG$ be an instance of a derivation scheme with $(O, E); O = O^0 \cup O^D$ where $O^0$ is a set of base objects and $O^D$ is the set of derived objects, and $E = \{ (u,v) \in O \times O : u \text{ used in the derivation of } v \}.$

**Definition:** Let $T_C \subseteq T = \{T^1, \ldots, T^n\}$. $T_C$ is a minset of $T$ if

1. $\bigcup_{t \in T_C} WS(t) = O^D$;
2. $\forall T^i, \forall T^j, T^i, T^j \in T_C, WS(T^j) \cap WS(T^i) = \emptyset$ (i \neq j)

**Definition** A schedule $S$ of transactions in $T_C \subseteq T$ is said to be minimal if it is consistent with the partial order $\preceq$ and $T_C$ is a minset of $T$.

Given a derivation scheme $DG$ and a set of transactions $T$, a minset of $T$ consists of a set of transactions that will derive the final set of objects without deriving any
object more than once. Thus if the cost of deriving every object is positive and finite, the minimal schedules are the optimal schedules for producing the final design objectives. In case of software configuration management, under nonuniform transaction management, the dependency between the various files can be expressed by a minset. However, given a set of transactions and the derivation scheme DG, there could be several minsets and hence several makefiles for obtaining the same final objective.

Example 1: Let DG be a derivation scheme as shown in Figure 3. The set of transactions associated with DG is \( T = \{T_1, \ldots, T_8\} \) where the read-set and the write-set of the transactions are shown in Figure 3. It can be seen that every element of \( \{T_2T_1T_5, T_1T_3T_5\} \times \{T_6T_8, T_7\} \) is a minimal schedule.

However, as the next theorem shows, determination of minset is NP complete (if \( |WS^i| \geq 3 \)). So unless \( P = NP \), a scheduler cannot efficiently schedule only those that are minimal.

**Theorem 2** Determination of minset is NP complete.

*Proof:* By reduction of 'minimum cover problem' [31].

Instance: \( DG = (O, E) : O \) is the set of objects and \( E \subseteq O \times O \). \( T = \{T_1, \ldots, T_n\} \) where \( WS(T^i) \subseteq O \).

Minimum cover problem is known to be NP complete (for \( |C| \geq 3 \)).

Instance: \( M \) = finite set of elements; \( C = \{c | c \subseteq M \} \)

Question: Is there a \( C' \subseteq C \) such that every member of \( E \) belongs to at least one member of \( C' \)?
Transformation: Let \( M = \{e_1, e_2, \ldots, e_k\} \) and \( C = \{c_1, \ldots, c_m\} \) \((| c_i | \geq 3)\). Let \( DG = (O, E) \) be a derivation scheme where \( O = M \cup \{e_0| e_0 \in O^0\} \) and \( E = \{(e_0, e)| e \in M\} \). Let \( T = \{T_1, \ldots, T_m\} \) be a set of transactions such that write-set of \( T_i = c_{m-i} \). Let \( S \) be a schedule of transactions in \( T \) and define \( C' \) such that if \( T_i \) is in \( S \) then \( c_i \in C \); If \( S \) is a minimal schedule then \( C' \) is the minimum cover.

Object Consistency Revisited

We will refine the notion of mutual consistency between objects for the nonuniform transaction model. Object \( O^i_j \) is said to directly derive object \( O^i_k \) (denoted \( O^i_j \rightarrow O^i_k \)) if the object \( O^i_j \) is read by the transaction that wrote the version \( O^i_k \). Object \( O^i_j \) is said to derive object \( O^i_k \) denoted \( O^i_j \rightarrow^* O^i_k \) iff (i) \( O^i_j \rightarrow O^i_k \) or (ii) \( O^i_j \rightarrow O^u_k \rightarrow O^i_k \).

Internal consistency is now redefined as follows:

(i) All instances of the initial objects are internally consistent.

(ii) An object \( O^i \) is internally consistent if the set of objects that derive it are internally consistent and pairwise mutually consistent.

We now extend the definition of mutual consistency to nonuniform transaction system as follows:

1. The initial set of objects \( \{O^{o1}, \ldots, O^{ok}\} \) are mutually consistent with every object \( O^i_j \).

2. Any two versions of an object are not mutually consistent: \( O^i_j \) is not mutually consistent with \( O^i_k \) if \( k \neq j \).

3. If \( \forall v, \forall u, u,v \in RS(T^i_j) \), \( v \) and \( u \) internally consistent and mutually consis-
tent \Rightarrow \forall p, \forall q, p, q \in WS(T_p^j) \text{ p and q are internally consistent and mutually consistent with } u, v.

4. If read-sets of two transactions are mutually consistent and write-sets do not intersect then the objects belonging to the write-sets are pairwise mutually consistent.

**Theorem 3** Let \( F \) be the set of final objects that are internally consistent and pairwise mutually consistent. Then \( F \) is derived by a minimal schedule.

**Proof:** (By Contradiction)

Let \( F = O_f^1, ..., O_f^k \) be the set of final objects that are internally consistent and mutually consistent. Let \( T_F \) to be a set of transactions such that:

(i) if \( O_f^i \in WS(T_j) \) then \( T_j \in T_F \) or

(ii) if \( O_f^k \in WS(T_i) \) and \( O_f^k \neq O_f^i \) then \( T_i \in T_F \).

Assume \( T_F \) is not a minset; from the definition of minset, we have two cases:

**Case 1:** \( \bigcup_{t \in T_f} WS(t) \neq O_f^D \). If this is the case, then the set of final objects \( F \) is derived without deriving all other objects in \( O_f^D \) -- a contradiction since DG is not redundant.

**Case 2:** \( \exists t_i \), \( \exists t_j \), \( t_i \), \( t_j \in F \) such that \( WS(t_i^i) \cap WS(t_j^j) \neq \emptyset \). Without loss of generality, let \( t_i \) and \( t_j \) create versions of object \( O_i^u \). From the definition of mutual consistency \( O_i^u \) is not mutually consistent with \( O_j^u \). Therefore if \( O_i^u \in F \), then \( F \) is
not mutually consistent - a contradiction since $F$ is given to be a set of mutually consistent objects. If $O^u \not\in F$ then since $t^i$ and $t^j \in T_F$, from the definition of $T_F$ we have:

$WS(t^i) \in O^{fl} | O^{fl} \in F$ and
$WS(t^j) \in O^{fm} | O^{fm} \in F$.

Since $WS(t^i) \cap WS(t^j) \neq \emptyset$, from the definition of mutual consistency, $WS(t^i)$ is not mutually consistent with $WS(t^j)$ and therefore $O^{fl}$ is not mutually consistent with $O^{fm}$ - a contradiction.

From Theorem 2 it follows that the determination of minset and hence a minimal schedule is NP-complete. Theorem 3 suggests that by requiring every transaction to use mutually compatible objects, the effect of a minimal schedule can be induced.

**Software Configuration Management**

Software configuration management (CM) deals with identification, organization, and control of modifications to software developed by a team of programmers. The task of coordinating the design activity of a large team involved with a software development is nontrivial. The main problems associated with CM are:

1. The double maintenance problem;
2. The lost update problem;
3. The interference or shared data problem;
4. The version control and revision control problem.
To avoid the double maintenance problem that might arise due to the presence of multiple copies of a software module, all modules are maintained at a common site. Careful 'check-out' and 'check-in' procedures are used to avoid lost update problems and logs are maintained and data abstraction techniques are used to avoid the problem with shared data. In environments where multiple versions need to be maintained, revision control schemes are necessary; maintenance of mutual consistency between the different modules becomes a nontrivial issue.

In Unix environments, the make program can be used to maintain the currency of object files. The data dependency between the various modules can be placed in a makefile, along with information for rebuilding the modules in case inconsistency is detected. The makefile can be stored under SCCS [69] or RCS control to provide automatic regeneration of the final object module.

In systems similar to those discussed above, the final module that is maintained current is unique. At any given time there is at most one set of mutually consistent data objects - the most recent version of the interface files, source code, and their object modules. However, a software design team may maintain several versions or 'releases' of software which are design alternatives, and not design revisions. Thus the most recent version is not the only correct version. Examples are software designed for different hardware configuration. Under these conditions, the tools discussed above do not provide adequate support. In addition, in the CM tools discussed earlier, it is assumed that there is only one way of obtaining a derived object from the set of dependent objects and this derivation scheme is specified in the makefile. As pointed
out earlier, such an assumption is not valid when design alternatives (design tools, processes, algorithms, etc.) are available and therefore the dependency graph of the final object is not unique. In such instances, internal consistency and mutual consistency can be used as correctness criteria for design objects. In the next section, we propose a mechanism for maintaining version control which uses such criteria.
A MECHANISM FOR VERSION CONTROL

In this section, we propose a mechanism for version control. We assume that the database supports multiple versions of each object. Let \( O = \{O^1, \ldots, O^n\} \) be the set of objects that are maintained by the database. Each object \( O^i \) is maintained as an object history \( OH^i \); thus the database is composed of object histories \( \{OH^1, \ldots, OH^n\} \).

Associated with every object history \( OH^i \) is an object manager \( OM^i \). The object manager implements the readmap and writemap functions, which provide support for the version control mechanism, and also provide the necessary concurrency control. Each object history \( OH^i \) is composed of a set of versions of the object \( O^i \) denoted by \( o_1^i, \ldots, o_j^i \). Object \( o_j^i \) is a tuple \((value_j^i, ilist_j^i)\), where \( value_j^i \) is the value associated with the object instance \( o_j^i \), and \( ilist_j^i \), informally, contains the ids of all objects that are involved in the derivation of the object \( o_j^i \).

Capability

The derivations are carried out by transactions. A transaction \( t_i \) can be uniquely identified by its id `cap_i`, where \( cap_i \) is drawn from a totally ordered set \( CAP \). The set \( CAP \) can be implemented by maintaining a system of synchronized clocks. Thus \( cap_i \) of a transaction \( t_i \) can be obtained by concatenating the initiation time of \( t_i \) with the id of the node where the transaction is invoked. This scheme gives system-wide unique values that are totally ordered.
**Definition:** An nc-pair is a set of tuples of the form (name, cap), where name ∈ O, the set of objects, and cap ∈ CAP.

Every object manager OM\(^i\) implements the function *readmap* which maps an nc-pair to an instance of an object or to null. The *writemap* function, also implemented by the object manager, places a version of an object in the object history.

**Definition:** ILIST is a set of nc-pairs with the following restriction:

\[ \text{Hist} \subseteq \text{ILIST} \text{ iff } \forall nc \forall nc' (nc, nc' \in \text{ilist}) \wedge \]

\[ (\text{nc.name} = \text{nc'.name}) \Rightarrow (\text{nc} = \text{nc'}) \]

That is, ilist is a partial mapping from name to cap.

Transactions have an *environment* associated with them. An environment is a 3-tuple (cap, ilist, oset) where cap ∈ CAP, ilist ∈ ILIST, and oset is a set of tuples (name, value). Initially when a transaction \( t^i \) is invoked, a cap\(^i\) is generated on behalf of transaction \( t^i \); the ilist\(^i\) and oset\(^i\) are initially empty.

Operations within transactions have to be controlled to guarantee version consistency. The ilist and cap that are associated with the transaction environments and object instances are used to enforce version consistency. A brief description of a scheme to enforce internal consistency of objects derived by a transaction is presented as protocol-VC below.

**Protocol - VC**

Let R and S be members of ILIST. We now define operation \( \text{vjoin}(R, S) \) denoted by \( R \Rightarrow S \) as follows:
If \( \forall r, r \in R \Rightarrow \exists s, s \in S \ s.t. \ r = s \) \lor \( \forall s, s \in S \Rightarrow \text{s.name} \neq \text{r.name} \) \land \\
\forall s, s \in S \Rightarrow \exists r, r \in R \ s.t. \ s = r \) \lor \( \forall r, r \in R \Rightarrow \text{r.name} \neq \text{s.name} \) \) \\
then \( R \leftarrow S = R \cup S \) \\
else \( R \leftarrow S = \emptyset \).

When a transaction \( t^i \) is invoked, an environment \(<\text{cap}_i, \text{ilist}_i = \emptyset, \text{oset}_i = \emptyset>\) is created. When a read request for an object \( x \) is made by \( t^i \), the nc-pair \((x, \text{cap}_i)\) is sent to the object manager \( \text{OM}_x \). The object manager \( \text{OM}_x \) now maps the request (through readmap) to an instance of \( x \), \( x_j \) where \( \text{cap}_j \geq \text{cap}_i \). If the request is permitted by the concurrency control mechanism at \( \text{OM}_x \) (e.g., if the object is not locked), then the value of \( x_j \) along with \( \text{ilist}_{x_j} \) is returned to \( t_i \). The read is said to be \textit{version consistent} iff \( \text{ilist}_i \leftarrow \text{ilist}_{x_j} \neq \emptyset \). If the read is version consistent, then \( \text{ilist}_i \) is updated to \( \text{ilist}_i \leftarrow \text{ilist}_{x_j} \). If the read is not version consistent, then the transaction can either abort or redo the read operation if backtracking is allowed. The vjoin can be performed by the object manager which has access to \( \text{ilist}_i \) and return a version that is consistent with the transaction’s \( \text{ilist}_i \) or ‘fail’ if no such version exists.

When the transaction \( t^i \) writes an object \( x \), the pair \((x, \text{value}_x)\) is placed in the \( \text{oset}_i \). When the transaction \( t^i \) commits, the following operations are performed.

1. For all \( \text{name}_j \in \text{oset}_i \) remove the nc-pair \((\text{name}_j, +)\) from \( \text{ilist}_i \).

2. For all \( \text{name}_j \in \text{oset}_i \), add the nc-pair \((\text{name}_j, \text{cap}_i)\) to \( \text{ilist}_i \).

3. For all objects \( \text{name}_j \in \text{oset}_i \) send “write(\text{name}_j, \text{ilist}_i, \text{value}_j)” to object manager \( \text{OM}_{\text{name}_j} \).

4. Send ‘release’ message to all object managers that have participated in the
transaction.

When the transaction $t'$ aborts, the transaction manager $TM_i$ associated with $t'$ sends "abort($t'$)" to all object managers that have participated in the transaction. This message may cause the OM's to release any locks that are held on behalf of the aborted transaction. The environment associated with the transaction along with the copies of versions saved in oset$_i$ is destroyed.

We now present the following theorem.

**Theorem 4** If every transaction in a schedule $S$ follows protocol-VC, then all objects written by transactions in $S$ are version consistent.

Informal proof: It can be seen that every "read" operation performed by a transaction under protocol-VC reads mutually compatible objects; consequently, the transactions write objects that are version consistent.

If the transactions can order their requests such that the objects that are farthest from the base objects (in the design scheme) are read before those that are closer, then unnecessary failures may be avoided. Furthermore, from the definition of vjoin operation it can be seen that the complexity of computing the vjoin is $\Theta(|O|)$, where $|O|$ is the size of the set of database objects, allowing an efficient implementation of the version control mechanism.
CONCLUSION

In this paper, we present two models of design transactions and formally presented the notion of version compatibility. Acyclic design schemes with a unique source (base object) and a unique sink (final object) can be modeled by the uniform transaction model. In this model, write-sets of transactions are unique. As an example, the set of VLSI CAD tools can be modeled as a uniform transaction model. When several design alternatives are available for each phase of the design, resulting in set of versions for each object type, coherent schedules maintain version consistency. This can be used as the correctness criterion in situations discussed, in the place of view serializability which has been used in conventional databases.

In the nonuniform model, acyclic design schemes can have more than one source and sink; the write-sets of the different transactions are non-empty. In this scheme, we show that determination of the set of transactions, and hence the schedule to derive non redundant set of derived objects is NP hard. We also show how internal consistency and mutual consistency of design objects can be used as correctness criteria for maintaining consistency of multiple versions of design objects. We also present a mechanism for maintaining on-line version control which can be used to generate correct schedules in the presence of concurrent design transactions. The proposed version control scheme is modular and can be implemented efficiently, independent of the concurrency control scheme.
Figure 1: Structure of a Silicon Compiler
Case (a): Sequential design schedules.

Case (b): Concurrent schedules with Objects renamed

Figure 2: Schedules for Design Transactions

Figure 3: A Derivation Scheme
PART III.

A DATA MODEL FOR DESIGN OBJECTS
ABSTRACT

Modeling concepts for handling structural data of hierarchically composed complex objects are presented. A complex object is modeled as an hierarchy of interfaces and an implementation. Objects sharing the same interface at any level have same attributes and attribute values up to that level in the hierarchy. This structuring facilitates storage of multiple object versions and their subsequent reuse in design databases, where multiple versions of design artifacts are routinely created and maintained. Other important concepts needed for design databases are presented, and in particular two types of instantiation - parametric instantiation, and conditional instantiation - are proposed.
INTRODUCTION

The Relational model (R-model) has been found inadequate for advanced database applications that are now expected of databases in areas such as CAD/CAM, office information systems, and software engineering. The inadequacy is primarily due to the following three reasons. First, conventional relational database systems with flat tuples do not provide adequate structuring mechanism to capture all information pertaining to an object. Second, even relatively simple applications in the non-traditional fields pointed out earlier, require computation of transitive closure. However, it is well known that transitive closure cannot be expressed by a relationally complete query language [3]. Third, simple tables of atomic components are not capable of capturing a large class of data encountered in such advanced database applications (e.g., text).

Attempts at solving the problems associated with the R-model can be classified into three groups:

1. Extensions to the R-model [22],[37].

2. Non conventional data-models including the Entity Relationship model (E-R model) [19] and works related to non-first normal forms(NEF²) [1], and more general models such as [7].

3. Object oriented approaches; efforts in this area are based on object oriented programming language paradigm. The distinguishing features of this paradigm are attribute inheritance, class hierarchy, and message passing [25],[46].
The first normal form (1NF) structuring constraint imposed by R-model forces a database designer to decompose complex objects into a set of homogeneous tuples, obeying the various dependency constraints. In the process the model loses the ability to express the schema in an easily understandable way. In NF\textsuperscript{2} models, the constraint that tuple attributes be atomic is relaxed allowing attributes to be relations. The R-model retains the structuring concept: the relations, with its operations extended to manipulate the complex structure. However, this approach also fails to capture the schema of the world of discourse in an easily understandable way.

In recent years, researchers have focused on object-oriented paradigm to model complex objects. Although several prototypes have been developed, it is unclear whether a user friendly interface can become available in the near future. Meanwhile the E-R model can be viewed as the first step in the direction of object-oriented models. E-R model offers the concept of entities (E) and entity sets to represent the objects of interest, along with relationship (R) and relationship sets, distinct from entities, to capture the association relating the entities. The third distinct concept is that of attributes which are distinct from entities and relationships, and are used to express the characteristics of the entities. The E-R modeling approach offers the designer a natural way for representing the 'world of interest'. However, current E-R models share the same inadequacy of R-model to represent complex objects.

A modeling technique based on E-R model for complex objects called molecular aggregation has been recently proposed [10]. Other modeling concepts needed for nested (composite) objects such as object interfaces, object versions, and instantiation
have been presented in a recent work [11]. Here, we extend the modeling techniques and propose a scheme suitable for design objects. In particular, we use VLSI and electronic circuit design as examples throughout to illustrate the concepts presented. In addition, we introduce the notion of parametric instantiation and conditional instantiation which will in most cases alleviate the problem of change propagation and change notification in design databases. While the primary focus of the proposed scheme is to encapsulate the structural information of a complex object, rather than its behavior, the scheme can be adopted for modeling geometric and structural data of other engineering artifacts as well.

The remainder of this paper is organized as follows. In the next section we briefly examine the requirements of a design database model. In Section 3 we present the concepts needed for representing complex objects and their versions. In Section 4 and Section 5 we present other concepts needed for modeling complex objects. Section 6 deals with a set of primitive operations for manipulating complex objects and Section 7 concludes the paper.
REQUIREMENTS OF A DESIGN DATA MODEL

In this section we present some of the issues that need to be resolved for an efficient implementation of a design database.

1. Complex Objects: The design artifacts usually are composed of a collection of heterogeneous records which together represent the design artifact - a complex object. The set of records cannot be conveniently represented by a small set of homogeneous records or tables of tuples as in the case of conventional database entities.

2. Composite Objects: Composite objects are complex design objects formed by integrating simpler objects that are defined independently, resulting in a composition hierarchy. The data model should facilitate the creation and maintenance of such a composition hierarchy. Note that all composite objects are complex objects as well.

3. Sharability of Objects: For practical considerations such as economy, efficiency of design effort and storage, composite objects are designed using components that are defined and available in a design database accessible for designers. The components in the library are shared by all designers. The model should facilitate such sharing. In addition, a composite object might use several instances of an object in its design. Therefore, during representation, such redundancy needs to be avoided.

4. Multiple Versions: Typical design processes involve creation and testing of multiple versions - both as design alternatives and as revisions to existing design.
Thus several versions may exist for each 'object type'. The data model should facilitate the creation and maintenance of such a composition hierarchy.

5. Abstraction Hierarchy: In a system where there are users at different levels with different views of the data, a hierarchy of data abstraction is essential for the users to access and manipulate the objects. In addition to the ease of use in handling and managing the complex data, an abstraction hierarchy allows for the stability of the system. Each abstraction level acts as a firewall for changes in the data as they evolve – which is often the case in a design database where all changes at lower level may not propagate up in the data hierarchy.

6. Operations on Objects: The operations on the design objects also pose some problems. The complex objects may have to be retrieved and manipulated as a whole or a part of the object may be modified. In both cases, the data model should facilitate the operations efficiently.

7. Multiple Representation: An object may have different interpretations in different 'views'; for example a circuit diagram could be viewed as a set of connectivity information or a graphic layout, depending on the environment.

8. Schema Changes: Unlike conventional databases, the schema of the design database changes frequently as the structure of the design objects change. Consequently, design databases should facilitate and support schema evolution.

In this paper, we address all the above issues except those concerning multiple representation and schema evolution. The problem of multiple representation is application dependent and needs to be handled during interface design.
COMPLEX OBJECTS

Design artifacts are complex objects and they can be represented by a set of heterogeneous tuples. The object can be abstracted and represented by a single tuple (with its unique object id among other attributes) at a higher level. The various attributes of the object can be represented by a set of inter-related tuples or tables of tuples at lower levels. Several proposals have been made to extend conventional database systems to include the notion of complex objects [56], [47], [11].

In a recent work [10], complex objects called molecular object are formed by the abstraction technique referred to as molecular aggregation. Molecular aggregation is the abstraction of a set of entities and their relationships into a higher level entity.

Using E-R modeling technique [19], a complex object can be reduced to a set of tables of relations. At the top level an object is denoted by a single tuple identified by its unique identifier (oid) in an object-table, along with a set of descriptive attributes OA₁,...,OAₙ. At the lower level, tables T₁,...,Tₘ hold other tuples that describe the object in finer detail. Tuples in table Tᵢ have key oid.ₖᵢ. See Figure 1 for an abstraction of a composite object gate which is made up of other objects (gates) and relations (wires). If the component entities in a complex object are themselves 'complex', then each of the components can again be further defined in terms of other simpler objects until all attributes are expressed in terms of a set of primitive objects. Figure 2 shows a gate ‘flip-flop’ which is made of two nand gates and a set of relations representing metal-runs or connections. The attributes of the flip-flop and its components can now be expressed by a set of tables: Tables(1-3).
Table 1: Gate

<table>
<thead>
<tr>
<th>Gate#</th>
<th>Desc</th>
<th>No.pins</th>
<th>Parent</th>
</tr>
</thead>
<tbody>
<tr>
<td>G1</td>
<td>2-nand</td>
<td>3</td>
<td>G3</td>
</tr>
<tr>
<td>G2</td>
<td>2-nand</td>
<td>3</td>
<td>G3</td>
</tr>
<tr>
<td>G3</td>
<td>flip-flop</td>
<td>4</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 2: Pins

<table>
<thead>
<tr>
<th>Gate#</th>
<th>Pin#</th>
<th>1/0</th>
</tr>
</thead>
<tbody>
<tr>
<td>G1</td>
<td>p1</td>
<td>I</td>
</tr>
<tr>
<td>G1</td>
<td>p2</td>
<td>I</td>
</tr>
<tr>
<td>G1</td>
<td>p3</td>
<td>O</td>
</tr>
<tr>
<td>G2</td>
<td>p1</td>
<td>I</td>
</tr>
<tr>
<td>G2</td>
<td>p2</td>
<td>I</td>
</tr>
<tr>
<td>G2</td>
<td>p3</td>
<td>O</td>
</tr>
<tr>
<td>G3</td>
<td>p1</td>
<td>I</td>
</tr>
<tr>
<td>G3</td>
<td>p2</td>
<td>I</td>
</tr>
<tr>
<td>G3</td>
<td>p3</td>
<td>O</td>
</tr>
<tr>
<td>G3</td>
<td>p4</td>
<td>O</td>
</tr>
</tbody>
</table>

Table 3: Links

<table>
<thead>
<tr>
<th>Wire#</th>
<th>From</th>
<th>To</th>
<th>Parent</th>
</tr>
</thead>
<tbody>
<tr>
<td>w1</td>
<td>G3</td>
<td>G1</td>
<td>G3</td>
</tr>
<tr>
<td>w2</td>
<td>G3</td>
<td>G2</td>
<td>G3</td>
</tr>
<tr>
<td>w3</td>
<td>G1</td>
<td>G3</td>
<td>G3</td>
</tr>
<tr>
<td>w4</td>
<td>G2</td>
<td>G3</td>
<td>G3</td>
</tr>
<tr>
<td>w5</td>
<td>G1</td>
<td>G3</td>
<td>G3</td>
</tr>
<tr>
<td>w6</td>
<td>G2</td>
<td>G1</td>
<td>G3</td>
</tr>
</tbody>
</table>
In the case of Figure 2, we assume that 2-nand gates and wires are primitive objects that need not be redefined further.
The scheme presented earlier was capable of capturing the complex relationship between different attributes of a complex object. However, such a scheme when used in a design environment will have several shortcomings. For example, if another version of an object implementation is designed (a flip-flop with one of its wires re-routed), then it would entail storing all the information associated with the new version as well, much of which is a repetition of the data associated with the previous version.

The second problem is in accessing the complex objects. The objects may be used in the design of other complex objects. In such cases, the information that is pertinent to the user are only a subset of the attributes of the object used. In general, the abstraction of the implementation is sufficient for a user to use a complex object. For example, in the case of integrated circuits, the pin configuration and electrical characteristics are sufficient for a designer to use a particular chip in a design. Therefore, it may be unnecessary for a designer to view the complex object in its entirety which contains details outside the scope of his interest. The projection of the implementation of the complex object can be captured by the interface of the object. The interface is the abstraction of the implementation, and the object interacts with the external world (other objects) through the interface.

Object interfaces can be viewed as the abstraction of the object implementation. Thus each object definition has two parts: the object interface and the object implementation. We now apply the concept of generalization abstraction to object
interfaces. In design databases, as we pointed out earlier, several versions of an object type may be designed (as design alternates or refinements). The different versions may have the same interface features and they differ only in the implementation specification. Thus the interfaces of different versions of the object may now be merged and represented by a single interface – an object which holds data common to the different versions of the object. In Figure 3, for example, two versions of flip-flop, each with its own interface and implementation are shown. Except for the routing of one of the wires, most of the features in both the objects are identical. Thus the common part of the attributes they share is extracted out and placed in the object ‘interface’ while the individual versions retain the attributes that are not shared. The association between the implementation and the interface can be maintained by the relation “version-of” which relates interface-types with its versions. In our work, this relation is denoted implicitly by including the column “Interface/Type” with the table denoting the entity “version”. It has to be noted that an interface of an object can itself be a complex object as it may have complex attributes. For example, the electrical characteristics of a gate can be represented by a set-valued attribute :“electrical-char.” This complex attribute can be represented by a table with a scheme (Char#, Char).

Hierarchy of Interfaces

An object can be represented as the sum of its two parts: its interface, and its implementation. The objects that share the same interface but different implementation can be grouped together and the interface information is generalized resulting
in a class of interfaces. All implementations of an interface inherit all information of the interface. If one of the attributes of the interface changes, then it results in a new interface (class). However, in a design database designers often create several versions of design objects as design alternates and as design revisions, possibly with different interfaces as well resulting in a profusion of interface types.

As pointed out in section 2, design data have to be represented in several levels of abstraction. In addition to the stability of the database, such structuring facilitates systematic database design, efficient implementation, and economy of storage.

Two well known forms of data abstraction are aggregation and generalization [73]. Aggregation abstraction referred to a form of abstraction where a relationship between a set of objects is viewed as a higher level object in which some of the details may be suppressed. For example the relationship between the entities Husband:John, Wife:Mary, License-no:1211 and Date:June 1, 1990 may be abstracted to ‘marriage’. In generalization abstraction a set of similar objects are grouped into a generic object where the members of the set are instances of the generic object. For example, a set of married couples can be abstracted to a generic object ‘couples’, where differences between individual members of the set are overlooked. The R-model exhibit both forms of abstractions. However the conventional R-model is insufficient to model complex objects.

The interface concept is extended to a hierarchy of abstractions, where at each level objects that share the same attribute values are grouped and represented as a single object at that level. Each class object (interface) inherits the attributes of the
class in the hierarchy above, of which it is a member. That is, class objects at each level is a generalization of the member objects.

**Attribute Types**

Entities are identified by the set of attributes associated with them. As in the case of R-model, the E-R model allows atomic valued attributes in the relations. We distinguish the following attribute types associated with an object. (1) Inherited attributes: attributes inherited from its parent class. These are non-modifiable. (2) Machine assigned attributes: fixed and non-modifiable (e.g., version number, creation date). (3) Environment dependent attributes: Attributes that are inherited from the environment. (e.g., size of font in a text, location of a cell in VLSI circuit). (4) Local attributes: These are version dependent attributes that are local to the version. The environment dependent attributes or parametric attributes can be associated with the interface resulting in a parametric interface or with a version resulting in a parametric version. In both cases, the domain of the parameter need to be specified. The attributes can then be used in the implementation as in the case of formal parameters in programming languages. This is further discussed in subsequent section.
Table 4: Gate Interface

<table>
<thead>
<tr>
<th>Type#</th>
<th>Desc</th>
<th>( a_1 \ldots a_n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>2-nand</td>
<td>...</td>
</tr>
<tr>
<td>T2</td>
<td>flip-flop</td>
<td>...</td>
</tr>
<tr>
<td>T3</td>
<td>2-nor</td>
<td>...</td>
</tr>
</tbody>
</table>

INSTANTIATION

Once an object is specified, then it can be used in the implementation of other objects by instantiating the object. Object instantiation allows for reusing or sharing of objects. An instance of an object inherits all the attributes of the object version along with the attributes specified in the interface. In addition multiple instances of the same object may be instantiated in the design of a single complex object. Instances of an object version may have their own instance specific attributes that are assigned during instantiation of the objects. The different instances are distinguished by associating an instance number along with the identifier sequence for the version number (interface name, version number). We assume that interface names are unique system-wide. Instances of object are related to the object implementation by a ‘Inst’ relationship which is in fact a ‘is-a’ relationship. The E-R diagram for a gate instance flip-flop is shown in Figure 4.

Tables 4-8 show the instance-relation for a flip-flop using two instances of a nand gate.
Table 5: Gate-version

<table>
<thead>
<tr>
<th>Type#</th>
<th>Version#</th>
<th>$b_1 \ldots b_m$</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>V1</td>
<td>...</td>
</tr>
<tr>
<td>T2</td>
<td>V1</td>
<td>...</td>
</tr>
<tr>
<td>T3</td>
<td>V2</td>
<td>...</td>
</tr>
</tbody>
</table>

Table 6: Table Gate-instance

<table>
<thead>
<tr>
<th>Type#</th>
<th>Version#</th>
<th>Instance#</th>
<th>parent</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>V1</td>
<td>gi1</td>
<td>T2 V1</td>
</tr>
<tr>
<td>T1</td>
<td>V1</td>
<td>gi2</td>
<td>T2 V1</td>
</tr>
<tr>
<td>T1</td>
<td>V1</td>
<td>gi1</td>
<td>T2 V2</td>
</tr>
<tr>
<td>T1</td>
<td>V1</td>
<td>gi2</td>
<td>T2 V2</td>
</tr>
</tbody>
</table>

Table 7: Pin-type

<table>
<thead>
<tr>
<th>Type#</th>
<th>Pin#</th>
<th>I/O</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>p1</td>
<td>I</td>
</tr>
<tr>
<td>T1</td>
<td>p2</td>
<td>I</td>
</tr>
<tr>
<td>T1</td>
<td>p3</td>
<td>O</td>
</tr>
<tr>
<td>T2</td>
<td>p1</td>
<td>I</td>
</tr>
<tr>
<td>T2</td>
<td>p2</td>
<td>I</td>
</tr>
<tr>
<td>T2</td>
<td>p3</td>
<td>O</td>
</tr>
<tr>
<td>T2</td>
<td>p4</td>
<td>O</td>
</tr>
<tr>
<td>..</td>
<td>..</td>
<td>..</td>
</tr>
</tbody>
</table>
Table 8: Links

<table>
<thead>
<tr>
<th>wire#</th>
<th>Parent</th>
<th>Type</th>
<th>Version</th>
<th>From</th>
<th>Type</th>
<th>Version</th>
<th>Inst</th>
<th>Pin</th>
<th>To</th>
<th>Type</th>
<th>Version</th>
<th>Inst</th>
<th>Pin</th>
</tr>
</thead>
<tbody>
<tr>
<td>w1</td>
<td>T2</td>
<td>V1</td>
<td>T2</td>
<td>V1</td>
<td>-</td>
<td>p1</td>
<td>T1</td>
<td>v1</td>
<td>gi1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>w2</td>
<td>T2</td>
<td>V1</td>
<td>T2</td>
<td>V1</td>
<td>-</td>
<td>p2</td>
<td>T1</td>
<td>v1</td>
<td>gi2</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>w3</td>
<td>T2</td>
<td>V1</td>
<td>T1</td>
<td>V1</td>
<td>gi1</td>
<td>p3</td>
<td>T2</td>
<td>V1</td>
<td>-</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>w4</td>
<td>T2</td>
<td>V1</td>
<td>T1</td>
<td>V1</td>
<td>gi2</td>
<td>p3</td>
<td>T2</td>
<td>V1</td>
<td>-</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>w5</td>
<td>T2</td>
<td>V1</td>
<td>T1</td>
<td>V1</td>
<td>gi1</td>
<td>p2</td>
<td>T1</td>
<td>V1</td>
<td>gi2</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>w6</td>
<td>T2</td>
<td>V1</td>
<td>T1</td>
<td>V1</td>
<td>gi2</td>
<td>p1</td>
<td>T1</td>
<td>V1</td>
<td>gi1</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Parametric Instantiation

Object interfaces and implementation can be specified using objects that are supplied as parameters during instantiation. If an interface has a parameter, then it is said to be a parametric interface. Every implementation of the interface inherits the parameter that is supplied to the interface during instantiation. The parameters of the object can also be expressed in tables.

An implementation can also be parametrized if the implementation uses parameters in the design. The parameters are supplied when a version is instantiated. Figure 5 shows the E-R model of a parametric implementation. The relation ‘Actu­als’ and ‘Formals’ capture the association between the various instances of the objects involved.

Figure 6 shows a specification of a flip-flop which uses two nand gates. The type of the parameters (interface) are specified while the actual version is left unspecified in the design. Users of the flip-flop implementation select a version of nand gate and create two instances of the nand gate and pass them as parameters to the instance of the flip-flop. This allows the users to create versions of objects that have required
characteristics by using a small set of 'template' versions and a set of base objects that are passed on as parameters. To capture the information in Figure 5 the set of tables is augmented by Tables 9–10.

**Conditional Instantiation and Lazy Evaluation**

In design environments, several design processes or activity may be in progress concurrently, co-operating to produce a final object. The final object is a composite of other complex objects, which may themselves be composite objects, thus resulting in a 'is-a-part-of' hierarchy. Designers of higher level objects (objects at the top of the composition hierarchy tree) have to delay their design until the lower level objects they intend to use in their design are specified. For example if a designer wishes to use an object of type G in his design, then he has to wait for the correct version of the object G (the one with attributes that matches his specification) to become
available so that a copy of the circuit can be instantiated in his design. This disallows concurrent design activity at all levels of the composition hierarchy.

Another problem with instances using explicit version name is the problem of inconsistency which arises when versions are modified or deleted. Due to the constant flux in the design databases, versions are periodically redesigned and new versions developed. Thus when composite objects are designed using instances of specific versions of other objects, if the object versions at lower level are modified then the composite object may not be consistent or correct. To maintain consistency of the composite objects, when an object O is modified, the changes have to be notified to other objects that use the object O in their design.

Let A be a composite object which has an instance of object B (refer to Figure 7). If the implementation of B is changed by its designer, then the design of object A may not be consistent if the design was based on some of the attribute values local to B. Two options have been suggested in literature for notification of change in design environments: flag based notification, and message based notification [20].

In both schemes, object A has to respond to the changes in B by verifying the design of A and revising the design if necessary. If the design constraints do not allow change in the attributes then object A becomes invalid. If the changes in B are within the design constraints of A, then the changes in B are reflected in the new design of A which has to be revised. This revision may result in changes in A's attributes, necessitating A to notify the changes to other objects that might use instances of A.
in their design.

The problems associated with modification of objects and change notification can be minimized or eliminated by using *conditional instantiation*. Under this scheme, objects can be instantiated without specifying the version explicitly. The instantiation is made by imposing constraints on the attribute values of the versions. Any version that meets the constraint can now be instantiated explicitly. Thus composite objects may be designed using objects that are still in the process of being implemented. The E-R diagram for conditional instantiation is shown in Figure 8.

The process of assigning an explicit version name to a conditional instantiation is called 'pinning'. When an object is pinned, the constraints are evaluated to identify the exact version as and when required by the system. This can be done by performing a select on the attributes involved in the condition over the domain of the object-types involved. By keeping the versions un-pinned, the implementations can remain current and unnecessary change notification/propagation messages and action can be avoided.
OPERATIONS ON COMPLEX OBJECTS

A set of primitive operations are presented below using which other complex operations can be built. The operations are classified into three groups: (1) operations on object interfaces, (2) operations on object versions, and (3) operations on object instances. The operations assume the availability of templates or storage area in the user space which holds the object that is manipulated. In each of the operations, the system accesses the various relations (tables) involved to retrieve/store the tuples that constitute the complex object. The object structure (schema) is maintained in a database which is accessed first to obtain the set of tables that hold the complex object.

Operations on Object Interfaces

The following operations are informally defined on object interfaces. The parameter ‘T-set’ refers to the set of interfaces (super type) of which the interface under discussion (type) is a member. The user space that holds the interface is called T-temp, and the T# refers to the unique identifier for the interface.

1. new-interface(T-set,T-temp): Install the new interface defined in T-temp in the set specified and returns the T# assigned to the interface.

2. read-interface(T#,T-temp): Read the specified object interface into T-temp.

3. delete-interface(T#): Delete the object interface from the corresponding I-set.

Note that the deletion of an object interface may create orphans out of the object versions that use the interface. Therefore, in the event of deletion of object interface, all object versions of the interface have to be deleted as well.
Operations on Object Versions

The operations allowed on object versions are listed below. The parameter V-temp refers to the area in the user space that holds the object version; V# is the machine assigned version number, and T# is the interface identifier.

1. create-version( T#,V-temp): Install the object version stored in V-temp as a version of interface T#; the call returns a version identifier V#.

2. read-version(T#,V#,V-temp): Read the specified version of the interface into the user area V-temp.

3. delete-version(T#,V#): Delete the named version from the set of versions of the object interface T#. As in the case of object interfaces, deletion of a version may require deletion of all instances of the version in the database.

4. update-version(T#,V#,V-temp): Replace the version V# with the new version in the V-temp. Composite objects that used the old version may now be inconsistent due to the changes in the version.

Operations on Instances

The following set of operations are defined for the object instances. The user area used for storing object instances is named I-area. The instance identifier is I#.

1. install-instance(T#,V#,I-area): Store the instance of the version that is in I-area. The call returns the identifier I# associated with the instance of the object.

2. read-instance(T#,V#,I#, I-area): Read the specified instance into the specified area.
3. **delete-instance**(T#, V#, l#): Delete the specified instance of the object from the database.

4. **pin**(T#, V#, l-area): Evaluate the conditions of instantiation of the object specified and instantiate the necessary parameters.
CONCLUSION

Modeling concepts necessary for modeling complex objects - particularly, schemes necessary for storing data associated with design objects are presented. We use examples from VLSI and electronic circuits to illustrate the concepts. We propose the concepts of parametric versions that facilitates the reusability of a set of design objects - the parametric versions/ interfaces to obtain several versions or instances of the object type but with significantly different characteristics. We also propose the concept of conditional instantiation which can be used to maintain integrity even in the presence of changes in object versions.

In addition to the set of primitive operations suggested, a query language capable of supporting complex operations need to be developed to support the model.
Figure 1: Molecular aggregation

Figure 2: Molecular aggregation of gate: Flip-Flop
Figure 3: Object implementation and interfaces
Figure 4: Object instantiation

Figure 5: Parametric implementation
Object P_flip-flop is_a T2
  version : V2
  local attributes: ..
  created-by : ..

  .... : ...

  components:
    FP1 : is_a T1;
    FP2 : is_a T1;

  relations:
    wire1: From: self.p1; To: FP1.p1
    wire2: From: self.p2; To: FP2.p2
    wire3:

end {object P.flip-flop}

Object counter is_a T4 { T4 is an interface for a binary counter}
  version : V1
  local attributes: ..
  created-by : ..

  components:
    gi3: T2.V2(T1.V1.gi1.T1.V1.gi2) {counter is the parent}
    {T2.V2 is a P.flip-flop ; T1.V1 is a 2-nand }

    ...

end {object counter}

Figure 6: A Parametric flip-flop
Figure 7: Change notification

Figure 8: Conditional instantiation
PART IV.

CONCURRENCY CONTROL IN DESIGN DATABASES
Transactions in design databases execute under an environment unlike conventional transactions. In this paper two critical issues relevant to the long-running design transactions are considered. A data model for organizing the design data and concurrency control mechanism for managing the transactions are proposed. The design (project) transactions are composed of subtransactions which are managed by the site where the project transactions are executed. While access to the design objects are controlled by the access protocol based on locking, they are not necessarily two-phased – only the access to the composite object hierarchy are serialized by following a tree protocol. The design database manages and controls the top level project transactions. A new type of transaction called the hypothetical transaction is proposed. These are used by the designers to explore alternate designs and may fail with high probability. Therefore, they are supported with minimal overhead. The hypothetical transactions can be upgraded to a regular transaction under certain situations.
INTRODUCTION

In this paper we propose a transaction management scheme for design databases, intended for engineering applications such as CAD/CAM, VLSI design, and software development. Transactions exhibit properties that are useful in design databases: object consistency in the presence of failures and interleaved concurrent transaction executions [33]. However, the general model of transactions as proposed by Gray and others cannot be directly applied to design operations or design transactions [8]. There are two serious limitations in applying the conventional transaction model which are developed for business applications, to design databases:

1. Type of data: Conventional databases are characterized by a small number of large files of homogeneous data records, whereas in design databases the design artifacts are composed of heterogeneous records.

2. Type of transactions: In conventional data processing applications, transactions run for short durations (seconds or minutes), while design processes are usually long lasting executing for several hours or days. This leads to reduced concurrency, increased probability of deadlocks and consequently, reduced throughput.

In this paper, we propose a model to organize design artifacts for efficient manipulation and also suggest a mechanism for managing transactions in design databases.
SYSTEM ARCHITECTURE

The design environment consists of a central database (also called public database) that is shared by all the users. The database is connected to a set of workstations and compute servers such as special purpose vector processors and graphics hardware, through a high speed local area network (LAN). Design transactions are executed on the workstations which have windowing facility and can concurrently execute several transactions. The workstations have their own local databases, which are used to store local data and also for caching design objects that have been checked out from the public database, as shown in Figure 1.

![Figure 1: The Design Environment](image)

Design transactions, initiated from the workstations, are modeled as a set of transactions, possibly nested, called project transactions. Each project transaction represents a set of activities that may be modeled by a set of conventional transactions. However, design processes need to co-operate and communicate with each other; they may need to interact in an arbitrary fashion. The conventional tree-like
nested hierarchy suggested by [60] is not sufficient to model such behavior; serializability requirement imposed on conventional transaction model may be unnecessarily restrictive in the design environment [34, 30, 57]. In this paper, we do not specify any concurrency control mechanism needed for the subtransactions of a project transaction. We merely point out that the conventional approach - serializability - is inadequate and other approaches such as those suggested [30, 57] will be necessary.

In this paper, we assume that a set of related activities operate under auspice of a project transaction. The transaction manager associated with the project transaction is responsible for managing the requests of all the subtransactions belonging to the project transaction. It is also the responsibility of the transaction manager to maintain the consistency of the data items that are manipulated by all the subtransactions belonging to the project transactions. The data objects in the public database are assumed to be consistent before the execution of a project transaction $T_{pi}$; after the execution of $T_{pi}$, the public database has to maintain its consistency, in the presence of other project transactions $T_{pj}... T_{pn}$.

Design objects that are used by the transactions are 'checked out' of the public database and placed in the local database where they can be accessed by all subtransactions that belong to the same project. The objects that are in the private database belonging to a project transaction are then 'checked-in' into the public database after the completion of the project transaction.

The design objects that are in the public database are shared by all the users
in the system. The project transactions on the workstations connected to the public
database access the shared design objects. Concurrency control has to be provided
to allow interleaved access to the shared objects by the project transactions. The
next section deals with the data model for the design objects. The mechanism for
concurrency control is provided in the subsequent sections.
DESIGN OBJECT MODEL

A design object consists of a set of attribute values and is treated as a single entity at some level of abstraction in a design environment. The attributes themselves may be atomic (elementary data types such as integer, character), or a set of other attribute values. For example, a VLSI circuit can be treated as a design object, which is in turn composed of other basic circuits. A complex design object may be designed in stages using simpler objects. A designer may design the basic building blocks which can then be used in the design of more complex objects. This approach, called bottom-up design, is natural and is prevalent in design environments. Thus the representation of design objects have to be stratified - allowing for access and manipulation of data at different levels of abstraction. Design objects cannot be represented by a simple scheme used in conventional databases such as the Relational model with tuples of atomic attributes (First Normal form or 1NF).

The inability of the Relational model, in its current form, to capture complex objects is one of the limitations of the model. Several extensions have been proposed to the Relational model [56, 22]. Researchers have also attempted to model complex objects using E-R modeling technique [19] in which the complex object can be decomposed into a hierarchy of relations [10]. Other attempts at modeling complex objects include non-first normal forms (NF^2) [2] and object-oriented approaches [24, 46]. These approaches tend to reduce the semantic gap between the data model and the world that is being modeled. In recent years object-oriented programming has been receiving considerable attention. Several object-oriented databases have either been built or under development [29, 46, 16]. Object-oriented approach appears to offer
some advantages over other models [58]. Although there is considerable experimental work in the area of object-oriented databases and the field is rapidly evolving, two important issues remain: (1) the lack of a common data model, and (2) the lack of formal foundation unlike the Relational model [4].

Object Interface and Implementation

We model design artifacts as complex objects - objects with complex attributes. The design object is composed of two parts - the object interface, and the object implementation. The interface of an object captures the common features of a set of object implementations and can be considered as an abstraction of the implementation. Objects that have different implementations but share an interface are called versions of the object. We associate a type with an interface. The attributes of an object are dependent on the implementation and the interface. The implementations have the attributes specified in their interface as well and the notion is similar to the inheritance property associated with objects in object-oriented systems (OOS). The concept of interface can be extended by defining an interface for a set of interfaces - extracting out the common features or attributes of a set of interfaces and placing them in another super-interface, resulting in a hierarchy of interfaces. While the modeling is akin to the class hierarchy in object-oriented systems it is not restricted to 'objects' in object-oriented systems. Complex object representation using tables of tuples have been proposed in [56, 37, 10].

Design of an object involves the specification of the implementation and its in-
terface. If a similar object exists in the database, then, only the implementation need to be designed as an interface of the object already exists in the database. Implementations of the same interface, also referred to as versions, are design alternates or revisions of previous implementations. When a version of an interface is designed and placed in the database, the new version inherits all the attributes that are exported by all the ancestors of the new version in the interface subtree.

Associated with every interface with descendants is a special descendant marked as the default interface or implementation. When only one descendant exists, then it is the default descendant. The latest descendant can be assigned as the default or alternatively the most stable node can be assigned as the default. During design process, objects may be selected either by (implicit) interface name alone or by (explicit) version name. Thus, when a descendant need to be selected by the parent, due to implicit naming, the default descendant is selected. This is further explained in the section under composite objects.

Instances

Once an object is designed, then instances of the object can be created and used. Instances of an object are related to the object versions (implementations) through the is-an-instance-of relation; it inherits all the attributes of the implementation and its interfaces. Several instances of an object may exist and can be distinguished by their instance/version/type names. Instances of an object may have some attributes that are local to the instances and are assigned during instantiation, and thus may differ from one another. For example, in the case of a VLSI design library, a version
of a gate may be fully specified; when the gate is used (instantiated) in some other
design, the instance object will be assigned its own X and Y co-ordinates in the new
design and may differ from another instance of the same gate version.

The interface, the implementations (versions), and the instances induce three
orthogonal hierarchies. The nested interface description of an object induces an
interface hierarchy- a rooted tree with leaves as object implementation. The arcs
along the tree correspond to the is-a relationship. The implementations of the object
results in a version hierarchy (in case of revisions, the versions hierarchy is referred to
as the history). The versions can be revisions of existing implementations, or design
alternates. The versions of an implementation are distinguished by version numbers.
The points on the instance-version plane (Figure 2) shows instances of a complex
object.

![Figure 2: Design Object Hierarchy](image-url)
Composite Objects

A complex object is an object with complex attributes (sets or tuples). In design environments, the design objects are often designed in a hierarchical fashion by first designing simple objects whose instances are used in the design of other complex design objects, resulting in another hierarchy - the composite object hierarchy. A composite object is a complex object which is composed of other complex objects which may or may not be composite objects. A composite object is specified through the object interface, and the composite object implementation. The implementation part consists of instances of the components and relations that are used to connect the components. The components are related to the composite object through is-a-part-of relation.

Examples of an interface, implementation and a composite object are given in Figures 3 and 4. In the interface description of the nand gate, the attributes 'author' and 'date' are local to the interface and are not inherited by the descendants of the interface (implementations and their instances). An implementation of the nand gate is object 'V1'; it inherits the attributes 'pins' and 'electrical-char' from its interface. In addition, the implementation has its own attributes (date, author, length, width etc.) Once an implementation is specified, then instances of the object can be created and used elsewhere. Each of the instances will inherit all the exported attributes of the implementation and its interface hierarchy.

Object P-flip-flop in Figure 4 and 5 is a composite object that uses two instances of the nand gate. One of the instances, NG1, is instantiated by a generic
instantiating template; it is instantiated without explicitly specifying the version of the implementation. The intention is that the designer of P-flip-flop is not concerned with the local attributes of the implementation of the nand gate. Therefore, the system selects a default version of the implementation (which may be the latest version of the implementation). In the case of the second instance, NG2, the version of the implementation that is used is explicitly named (V1). If generic templates are used, the local attributes of the specific implementation are hidden from the composite and are not available to the composite object. For example, the attribute electrical-char2 associated with the default version is not seen by the composite object P-flip-flop.

The complex attribute 'components' is set valued -set of gate instances- elements of which are complex objects. Every instance of the complex object P-flip-flop creates two new instances of nand gate. The instances of the components can be traced from the parent object through their instance numbers assigned during instantiation. The design objects are placed in a central database shared in a convenient form. Concurrency control aspects are discussed in the subsequent sections.
Interface nand is a gate {interface 'gate' has been specified}
%author:... {local attribute of the interface}
%date:...
no-pins: {all versions of the gate have the same number of pins}
pins:
  p1:... {input/output, the location,...}
  p2:..
  ..
electrical-char: ..
  ..
end {interface nand}

Implementation V1 is a nand.gate
  %author
  %date
  length:..
  width:..
  elect-char2:..
  .....
end {implementation V1}

Figure 3: Example of an interface and an Implementation
Implementation P-flip-flop is_a INT-FF {INT-FF is the interface }
%author: ...
%date: ....

components: {NG1 and NG2 are instances of nand gates }
  NG1 : is_instance nand.gate
      {NG1 is an instance of default version of nand}
  NG2 : is_instance V1.nand.gate
      {NG2 is an instance of version V1.nand}

{Wires are modeled as relations so that an existential constraint can be placed on them}

relations:
  wire1: From: self.pins.p1; To: NG1.pins.p1
  wire2: From: self.pins.p2; To: NG2.pins.p2
  wire3:...
  .......
end {object P-flip-flop}

Figure 4: Example of a composite object
Figure 5: Interfaces, Implementation, Instances and Composite Objects
OPERATIONS ON COMPLEX OBJECTS

Project transactions issue queries to the public database to request access to the nodes in the object hierarchy. The nodes in the object hierarchy can be an interface, an implementation, or an instance of an object. The set of operations on the nodes are:

1. Read interface: The attributes of an interface may be read through a query of the form “Select all interfaces of ... such that ...”.

2. Modify interfaces: The attributes of the interface may be modified by a query of the form “Update the attributes of... as ...”.

3. Add new interface: A new interface can be added to the database by a query of the form “Place the interface... under ....”.

4. Read implementation: A transaction may read the attributes of an implementation through a query of the form “Select implementations of ... such that ....”.

5. Modify implementation: The attributes that are local to an implementation may be modified by a query of the form “Update the implementation of... as ....”.

6. Add new implementations: New versions of an object implementation may be added by a query of the form “Place the implementation.... as a version of ...”.

7. Read instances: The attributes of instances can be read by a query of the form “Select instances of.... such that ....”.
8. Modify instances: The local attributes of an instance can be modified by a query "Update the instances of ... such that ....".

9. Add instances: New instances of an object can be added to the database through queries of the form "Place the instance .... as an instance of ...".

10. Delete Implementation: An implementation can be selected and removed from the database by a query of the form "Delete implementation ... ".

11. Delete Instances: A set of instances can be selected and deleted from the database by a query of form "Delete instances of .... such that .....".

In addition to the operations discussed above, composite objects can also be accessed to be read or modified. The discussion of operations on composite objects is deferred.
CONCURRENCY CONTROL

When a node in the object hierarchy is selected for access either in the shared mode or exclusive mode a lock is placed on the object in the public database and a copy of the object is sent to the site where the transaction is invoked. This model of computation is called the fixed action model, where the data objects migrate to the site where the transaction is executing. In conventional databases, the transaction model is called the fixed object model; transactions when accessing data residing at different sites create subtransactions that migrate to the site of the data object. In both schemes, concurrency control has to be maintained to ensure consistency of data objects. The protocol necessary for accessing the objects are discussed below.

Lock Protocol

All project transactions maintain check-out and check-in procedures. The transactions check-out objects that need to be manipulated - locks are placed on the object in appropriate mode, a copy of the object is made and sent to the transaction manager requesting the object. After completion of the design process, the transaction checks-in the data. A transaction can check-in the data at any time during its execution. Once a data object is checked-in, then it becomes available for other transactions to check-out. Thus, if a project transaction aborts after checking-in some of the design objects it created, then it may not undo the effects of the transaction fully. That is, only level two consistency criteria is satisfied as described in [34]. We require project transactions to be units of consistency and not necessarily units of recoverability. By not requiring two phased locking (check-out/check-in) and by allowing access to objects that are checked-in by transactions that have not yet committed, we do not
force transactions to wait for other transactions to commit. When a transaction fails, this can avoid cascaded aborts.

**Locking Modes**

Whenever shared objects are to be accessed by different transactions concurrently, it is necessary to control access to the shared objects in order to maintain object consistency. Two modes of locking are prevalent: shared mode lock in which, multiple transactions can have the object locked in shared mode, and exclusive mode lock where only one transaction is allowed to lock the object in exclusive mode. In the object hierarchy, the nodes are related to their parent by the 'is-a' relation. Thus, all nodes of a rooted tree can be treated as a part of a single relation. When a node in the object hierarchy is accessed by a transaction, it is necessary to ensure that the components that are related to the node are not modified by other transactions that are executing concurrently. To this end, the relevant parts of the object hierarchy need to be locked in appropriate mode. When nodes of the object hierarchy are locked other issues have to be considered. They are summarized below.

1. The number of locks that are placed to access an object need to be minimized to improve the efficiency of the system.

2. The locking scheme should not be unnecessarily restrictive; when possible transactions should be allowed concurrent access.

3. The scheme should be simple to implement.

An object may be locked in one of the following modes:

(1) Read lock (R).
(2) Write lock (W),
(3) Modify lock (M),
(4) Append lock (A),
(5) Outdated lock (OD),
(6) Intend Read lock (IR), and
(7) Intend Write lock (IW).

Intend read lock (IR), intend write lock (IW) and append locks (A) can be applied only on object interfaces.

**Read Interface**: When an interface \( \text{Int} \) has to be read, a read lock is placed on the interface and *all* the nodes belonging to the subtree rooted at \( \text{Int} \) down to the implementation are locked in IR mode. The instances belonging to the subtree are not locked. When a set of interfaces are to be locked, then the node that is the common parent of the interfaces to be selected is locked in Intend Read mode. The nodes of the subtree locked in IR mode may now be selected and locked in R mode.

**Write Interface**: An object is write locked if it has to be modified. When an interface has to be modified, two options are available. The first option updates the interface and overwrites the interface; the descendants of the interface (other interfaces, implementations and instances) now may become inconsistent with the interface that is being modified and may need to be recompiled or redesigned. If this is the case, then explicit Modify locks (M lock) are issued on all the descendants of the interface (interfaces and implementations only). The time of last modification is updated to reflect the modification.
In the second case, modification of the interface results in the creation of a new version of the interface; the previous version is not overwritten. However, there is a subtle difference between creation of a new version and revision. When a totally new version of an interface is created in a design scheme, both the new and the old versions co-exist in the design database. The new version is treated as a design alternate or another 'model' of the same family. Both versions become available for use in the design process; implementation and instances can be designed using either versions. However, in the case of revision, while both interfaces co-exist, no new nodes can be added to the subtree under the node that has been modified. All the nodes of the subtree under the node that has undergone modification are locked in OD mode. The OD lock is maintained by a single bit associated with the objects. Figure 6 shows the two cases.

Select a set of Interfaces/Implementations: Intend read and Intend write locks are used to lock an interface node when several of its immediate descendents may be selected and locked in either Read or Write mode. That is, an implicit Read/Write lock is issued on the immediate descendents. This is done to avoid unnecessary rollback or aborting of transactions. In schemes involving hierarchically organized data objects, [32] locks are propagated to the root. However, we envision that in a design database, most of the operations would involve data objects close to the leaves of the tree hierarchy. Consequently, the proposed scheme offers the advantage of using less locks. Meanwhile, it also offers a compromise by propagating the lock to the parent so that conflicts can be detected earlier.
Add an Interface/Implementation: When a new node is to be added to an interface of an implementation, the interface or the implementation has to be append locked. Append lock prevents other transactions from modifying the interface. In addition, adding of a new descendent node may necessitate some modifications to the local attributes of the parent (e.g., the latest version number). Therefore, the append operations have to be serialized.

Recompile an Interface/Implementation: When an object has been marked with modified lock, it has to be recompiled or redesigned to make it consistent with the modified ancestor. The compilation process has to start at the node immediately below the node that was last written into (the one that underwent the latest modification). That is, a write lock is granted to a node whose parent is not locked in modified mode. The subtree that is affected can now be progressively updated from top-to-bottom order to reflect the change. The modified lock is maintained by a single bit associated with an object. The bit is set when an object above it in the hierarchy is written into. Timestamps associated with the object are used to maintain and clear the modified lock. When the last-modified-time of the descendant $TM_{des} \leq TM_{any}$, the last-modified-time of the parent, the modified lock is cleared. Also, note that when new descendents are added to an object, only the local attributes are modified and therefore, the last-modified-time is unaffected.

Read/Write Implementation: When an implementation is read or written, its instances in the hierarchy tree are not locked in any mode. However, it is necessary to verify that the parent interface of the implementation has not been
locked in modified mode. If an implementation is modified, then as in the case of interfaces two options are available: over-write the old interface setting the modified lock and the last-modified-time, and creation of a new version, setting outdated lock on the previous version. The effect of these locks on instances are discussed when operations on instances are explained.

When modifications to a default implementation is made, the instances that are direct instances of the implementation are modified locked or locked in outdated mode. However, in the case of revision, the instances that are indirect descendants are assigned to the new version of the implementation (which now becomes the default version); in the case of over-write, the instances are unaffected.

Delete Implementation When an implementation is deleted, all direct instances of the interface are locked with M-lock and their parent is set to null. Thus when the instances are accessed, the modified lock along with null parent indicate that the instances are not valid. If the deleted implementation is the default version for the interface above, then a new default is assigned to the interface, and all indirect instances that were the descendants of the implementation that is deleted are assigned to the new interface.

Read Instances When an instance of an implementation has to be read, read lock is set on the implementation of which it is an instance. However, if the last-modified-time of the implementation is more recent than the last-modified-time
of the instance, the read lock is not allowed.

**Write Instances** : When an instance has to be modified, the implementation that is associated with the instance is locked in intend write mode. If the interface has been locked in outdated mode, than the write locks are disallowed. The instance selected to be modified is locked in write mode and the object can be checked-out. When the instance is checked-in after modification, if the instance had been marked with modified lock, it is cleared. Only the local attributes are allowed to be modified; attributes that are inherited from the implementation/interface hierarchy cannot be modified.

**Delete Instances** If an instance is to be deleted, then there are two cases to consider. If the deleted instance is an independent object and not a part of a composite object, then the set of tuples corresponding to the instance are removed. If the instance is a composite object, then its components need to be deleted too. Since, there could be a large composite object hierarchy, all instances of all the components belonging to the hierarchy are not recursively deleted immediately. The immediate component instances are marked 'deleted'. The object itself is deleted when all its components are actually deleted.

The compatibility matrix of the locks are show in Table 1 where a \(\checkmark\) is used to denote compatibility.

**Lock Upgrade**

It may be necessary for a transaction to first acquire a lock in a particular mode and then later decide to upgrade the lock to a more restrictive mode. For example,
Figure 6: Modification of an Object

Table 1: Compatibility Matrix

<table>
<thead>
<tr>
<th></th>
<th>RL</th>
<th>WL</th>
<th>IR</th>
<th>IW</th>
<th>M</th>
<th>A</th>
<th>OD</th>
</tr>
</thead>
<tbody>
<tr>
<td>RL</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>WL</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>IR</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>IW</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>M</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>A</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>✓</td>
</tr>
<tr>
<td>OD</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
a transaction may first acquire a write lock on an object interface. This results in modified locks being placed on all its descendents. The transaction can request write locks on the descendents which have been previously locked in modified mode. In this case the write locks are also granted on the descendents.

The operations that have been defined on the complex objects are on objects that were assumed to be independent - not made up of other complex objects. In case of composite objects, other considerations arise. These are discussed in the next section.

Operations on Composite Objects

In the case of non-composite objects, object instances are only related to their implementation through the is-an-instance-of relation, which has attribute inheritance property that has been discussed earlier. Composite object instances inherit the attributes of their interface hierarchy, and the implementation. The implementation of a composite object is composed of local components and instances of other complex objects (relations, components in Figure 4). That is, in the case of composite objects, an instance of an object can be related to an instance of another composite object through a 'is-a-part-of' relation. While the local part of a composite object is accessible only through the composite object implementation, the components are accessible through the implementation/interface hierarchy associated with the components. Therefore, it is necessary to ensure that two different transactions do not access the object instances through the two different hierarchies in conflicting modes leading to an inconsistent state.
Reading Composite Objects  Let a composite object $O_a$ be composed of instances of objects $O_b$ and $O_c$, then when an instance of the object $O_a$ is read, it is necessary to maintain the consistency of the components $O_b$ and $O_c$. That is, when a transaction is accessing an instance of $O_a$, no other transaction is allowed to affect the attributes of instances $O_b$ and $O_c$. This is achieved by locking the implementations of the components of the composite object in Read mode. Locking of implementations of the components will permit other transactions to read the instances of the components, but will prohibit modifications to the interfaces, implementation and instances of the components. When a composite object instance is locked in write mode, then the component instances are locked in write mode and their implementation is locked in read mode. This prevents simultaneous updates of the components through the different hierarchies.

In addition, we need to ensure that the composite object $O_a$ is still current. Since the design of the composite object $O_a$, the components $O_b$ and $O_c$ may have been modified. Therefore the components of the composite objects have to be inspected. Also, if any part of the instantiating template has a modified lock set, then the object has to be recompiled. However, if the instantiation is through generic template, (such as NG1 in Figures 4 and 5), then it may not be necessary to recompile the instance.

Modification to Components  When interfaces or implementations of an object that are used in the design of a composite object are modified, then it may be necessary to redesign or recompile the composite object. As in the case of write on interfaces, two alternatives are available: (1) setting of M lock, and (2) setting
of outdated lock. In the case of M lock on instances of components, the composite object instance has to be rewritten to reflect the changes. However, if an OD lock is set, then the composite object is allowed to exist; but no new instances of the composite object are allowed.

If the changes are made only on the implementation (as is often the case in design environments), then the situation is different. If the instances of the components are instantiated through the interfaces (generic instantiating templates), then the changes to the implementation do not affect the composite object. Under such circumstances, the instances of the components are not marked with M locks.

Atomic Subtrees (AST)

It has been pointed out that objects in the interface/implementation hierarchy can be locked and released by project transactions in an arbitrary fashion, as long as the locking protocols are followed. That is, no effort at maintaining serializability of project transactions is attempted. The argument is that the project transactions themselves are composed of transactions that change the database state from one consistent state to another. In case of nested transactions, the concurrency control is the responsibility of the transaction manager and the local database manager associated with the local database on the workstations where the design objects are cached. If an object is checked-in by a transaction $T_i$, then it can be checked-out by another transaction $T_j$, even before $T_i$ commits. Also, after checking-in an object, a transaction can check-out another complex object. That is, project transactions do not follow two-phased locking protocol.
However, in the case of composite object hierarchy, this cannot be applied. For example, consider the composite object $O_a$, composed of instances $O_b$ and $O_c$. The objects are interdependent; modifications to the composite object $O_a$ affect the components $O_b$ and $O_c$. Consider the following sequence of events involving transactions $T_1$ and $T_2$.

1. $T_1$ reads components $O_b$ and $O_c$.
2. $T_2$ modifies $O_a$ and checks-in $O_a$.
3. $T_2$ checks-out objects $O_b$ and $O_c$.
4. $T_1$ reads the new version of object $O_a$.

Transaction $T_1$ will find the composite object to be inconsistent with the components. Thus, the composite object subtree has to be treated as a single object and operations on its components should be atomic. Objects that do not belong to the same composite hierarchy tree can be treated independently. Thus, the composite object hierarchy induces a partition on the set of nodes in the hierarchy. The set of nodes belonging to a composite object tree forms an atomic object and the subtree is called an atomic subtree (AST). A similar notion, applicable to conventional databases, referred to as atomic data sets has been proposed in [71]. It can be shown that if access to ASTs in a database are serializable, then the database remains in a consistent state. Therefore we require transactions that access the composite objects to be serialized. The protocol for accessing composite object is given below:
Protocol for Accessing Composite Objects

Let $O = (o_1, \ldots, o_n)$ be the data entities. Without loss of generality, assume that the data entities are arranged in a linear order $o_1 < o_2 < \ldots < o_n$. If the transactions are allowed access to the entities only in the order they are arranged, then the transactions can be serialized. This is called Path Protocol [62]. Now, note that the nodes of the atomic subtrees are arranged in the order induced by the parent-child relationship. A generalization of the path protocol called Tree Protocol [62, 72] can be applied to the atomic subtrees. The protocol for the atomic subtrees is presented below.

1. Any node of a composite object subtree can be locked first.

2. If two nodes of a subtree are to be locked, then their common ancestor has to be locked (locking protocol of complex object still hold).

3. Locks can be placed on the tree from root-to-leaf order only.

4. If a lock on a node is released, then it cannot be locked again.

The protocol ensures serializability of transactions that access an ADS. The object manager associated with the public database maintains the lock and enforces the protocols.
HYPOTHETICAL TRANSACTIONS

In design databases, transactions do not necessarily create new design object that are used by other transactions. Often, designers try out new designs in an attempt to test out new or different schemes or ideas. The results of such transactions are never observed by other project transactions. The objects that result from such transactions need not be stored in the database permanently. One could consider such transactions as those transactions that always abort. We call such transactions hypothetical transactions. Hypothetical transactions have also been discussed by Garza and Kim in [32]. In their scheme, the hypothetical transactions always abort. Whereas, we allow a transaction that started as a hypothetical transaction to upgrade its status to a regular transaction whose effects can be observed by other transactions.

Hypothetical Locks

When a hypothetical transaction \(HT_i\) is initiated, it is assigned a time-stamp \((ts_{ai}\) which acts as a unique identifier. Unique names can be established for every project transactions by the scheme suggested by Reed in [67]. All lock requests of hypothetical transactions are immediately granted. The data manager keeps track of the locks on the objects and the transaction that requested the lock. The type of locks available for hypothetical transactions are: \(H\)-read (\(H\text{-read}\)), \(H\)-write (\(H\text{-write}\)), \(H\)-append (\(H\text{-append}\)), \(H\)-modify (\(H\text{-modify}\)), and \(H\)-outdated (\(H\text{-outdated}\)). The locks do not conflict with any other locks issued to other transactions and therefore they are always granted. That is, hypothetical transactions act as if they are executing alone.
However, within the context of a hypothetical transaction, the operations follow the lock protocols described earlier. For example, if an interface $In$ is modified and if $In$'s descendents are locked in H-modified mode, then the hypothetical transaction cannot request H-read lock on any of $In$'s descendents.

When a hypothetical transaction commits (terminates), all locks are released and all objects that were created by the hypothetical transaction are discarded. However, during the course of the hypothetical transaction, the designer may find the result of the design process acceptable, and may wish to make the effects of the transaction permanent. That is, the hypothetical transaction is to be made into a regular project transaction. The protocol necessary for such a conversion is discussed next.

**Lock Upgrade**

When a hypothetical transaction $HT_1$ decides to upgrade to a regular project transaction, it first sends an $Upgrade(id)$ request to the public database’s object manager, where $id$ is the unique identifier of the transaction requesting the upgrade. Note that the unique id can be formed by concatenating a sufficiently long bit sequence representing the start time of the transaction with the site number. Let the time-stamp associated with the hypothetical transaction $HT_1$ be $ts_s$ and let the time-stamp associated with the Upgrade request be $ts_f$. The Upgrade request of the hypothetical transaction $HT_1$, can be allowed or disallowed. If the request is allowed, $HT_1$ will be upgraded to a regular transaction and is allowed to commit, making permanent the changes it has made to the public database. It is now necessary to place $HT_1$ in
To serialize the hypothetical transaction, it is necessary to ensure that by upgrading HT$^1$ to a regular transaction, (1) the transactions that have been committed are not affected, and (2) other regular project transactions that are currently executing are unaffected.

The protocol can be informally explained using the following scenario. Let $T_i$ be a transaction that was active at $t_{s_1}$, the start time of the hypothetical transaction HT$^1$. Let the set of objects that were locked in W-mode, M-mode or OD-mode by $T_i$ be $O_{w_1}$. If transaction HT$^1$ was invoked as a regular project transaction, then it would not have read or written any of the objects $\in O_i$ until after $T_i$ committed or checked-in the objects. Similarly, let the set of objects that were locked in W-mode, M-mode or OD-mode by HT$^1$ be $O_{wH1}$, and the set of objects that are locked by HT$^1$ in R-mode and W-mode be $O_{rH1}$. Let $T_j$ be a transaction that has started before $t_{s_1}$. Then it follows that, if HT$^1$ had been a regular transaction, then $T_j$ would not have locked any object $\in O_{wH1}$ in R-mode or W-mode, until after the
objects were released by HT\(^1\). Let the set of objects that are locked by \(T_j\) in R-mode or W-mode be \(O_{rtj}\). Then, it follows that if \(O_{wti} \cap O_{RH1} = \emptyset\), HT\(^1\) will not affect the transaction \(T_i\) (see Figure 7). Similarly, if \(O_{rH1} \cap O_{wtj} = \emptyset\), then allowing the upgrade will not affect other transactions that are executing. Note that it is not necessary to compute the set \(O_{wti}\). It is sufficient to compute the set \(O_{rH1}\), and verify that none of the objects have been written by a transaction after \(t_{ss}\). In case of a conflict, the upgrade request is denied and the hypothetical transaction is allowed to abort.
DEADLOCK AND RECOVERY

The design database may fail due to hardware failures, software module failure or other events such as power interruptions. In our discussion we consider only failures due to transaction aborts. We treat the failure of the site associated with a transaction \( t_i \) as the failure of the transaction \( t_i \). Transactions can abort due to site failures (as discussed above), communication failures, software errors or due to deadlocks. Two transactions \( t_i \) and \( t_j \) are deadlocked if \( t_i \) waits for \( t_j \) to release an object A, while \( t_j \) waits for \( t_i \) to release an object B. In dynamic locking schemes, deadlock probability is proportional to the fourth power of the transaction execution time [35]. Thus, considering the long duration of design transactions, deadlock probability is considerably higher.

In our environment, the design objects are arranged hierarchically, and follow the path/tree protocol in the case of composite objects. The tree protocol is free from deadlocks [62]. Consequently transactions accessing the composite object hierarchy are never deadlocked. Even in the case of the interface/implementation hierarchy, one would expect transactions to restrict access to a subtree of the hierarchy, thereby eliminating the problem of deadlock. However, it is possible for transactions to access objects in a conflicting fashion resulting in a deadlock. In such an event, a deadlock detection mechanism is necessary to resolve the deadlock. The lock-manager will keep track of the lock requests of transactions, and in case of inordinate wait, can invoke one of the deadlock schemes suggested in [48]. When a deadlock is detected, one or more transactions involved in the deadlock may be aborted. In this case, the locks are reset and the database has to be recovered to a consistent state.
Recovery of data objects are achieved by *shadowing*; when write locks are requested (when objects are checked-out), a copy of the object is made and the copy is sent to the node requesting the object and the original object is retained. Thus, abortion of a transaction involves merely removal of all locks issued by the transaction. In case the aborted transaction has already checked-in new versions of objects, they are retained. As we have pointed out earlier, only consistent objects are checked-in by transactions, and therefore they need not be undone. That is, project transactions are not units of recovery; the local site maintains sufficient log information to restart the project transaction from the point of failure, if necessary. We place, on the project transaction manager, the responsibility of logging and checking of consistency of objects that are checked-in. This allows the central database to provide a high degree of availability and concurrency while allowing the project transactions to provide their own concurrency control mechanism that may be necessary.
CONCLUSION

In this paper, we presented a scheme for organizing complex data objects in a form suitable for design databases. By stratifying design data into a hierarchy of interfaces and implementation, we allow for concurrent access to different parts of the data, while avoiding representation redundancy and unnecessary change propagation or redesigning of objects. A scheme for managing long-running design transactions (project transactions) on the public database is also presented. Furthermore, hypothetical transactions – a design transaction that generally aborts – is supported by the scheme. Protocol for upgrading the hypothetical transaction to a regular transaction is also provided.

Composite objects are grouped into object called Atomic Subtrees. (AST). Elements of an AST are inter-related and therefore, to maintain consistency of the composite objects, operations on the elements of the set have to be atomic. A protocol for accessing and manipulating composite objects is also presented.
PART V.

TRANSACTION MANAGEMENT IN DESIGN DATABASES
ABSTRACT

Conventional database systems are not suitable for handling advanced applications encountered in engineering such as CAD/CAM, CASE, CAE, and VLSI design. The databases in such environments, also called design databases, are characterized by the presence of large number of complex data objects denoted by a large number of small tables, as opposed to a few large tables encountered in conventional databases. The transaction model used in tracking databases for banking, inventory control and other such applications use view-serializability as the correctness criteria. While view-serializability is appropriate for tracking databases, it is unnecessarily restrictive for engineering databases. We propose a transaction model that is suitable for accessing shared design data based on co-serializability. The transaction model supports long duration transactions with intermediate commit points. In addition to the conventional nested transaction hierarchy, the model allows for co-operation between nested design transactions that is needed in a design environment.
INTRODUCTION

Advanced database applications such as those encountered in CAD/CAM, VLSI Design, CASE and office automation require new type of control and management. The conventional transaction model proposed for data processing are not adequate for these applications.

In the conventional model, transactions are modeled as atomic actions; the entire transaction commits and the updates are made permanent, or it aborts and the effects of the transaction are undone. The intermediate state of the database is not visible to other transactions. It is assumed that a transaction, in the absence of concurrency, takes the database from one consistent state to another consistent state. Therefore, conventional transaction systems maintain database integrity by allowing only those transaction schedules that are serializable.

Throughout this paper, we shall refer to the transactions encountered in the non-conventional applications as design transactions. Serializability requirement for transaction schedules is too restrictive for design transactions. For example, consider the following issues:

- The design transactions are often computation intensive and are expensive to execute (in terms of resources used). Thus, 'abort-and-retry' procedures used in conventional schemes are not economical.

- Design projects (and other applications mentioned earlier) often produce 'partial results' that may be correct (and therefore used later), while the objective
of the project itself may not be reached. Under conventional schemes, if the project has been under a transaction scheme, the partial results will be discarded if the transaction (project) aborts (fails). Thus, the 'all-or-nothing' model required by conventional models are not suitable for design transactions.

- While serializability requirements are sufficient condition for database consistency in conventional databases, they are too strong for design environment. In a design environment, several design transactions may be active concurrently, each attempting to do similar work. Several 'versions' or design alternates may be developed concurrently, all of which may be used by other design processes. Thus, multiple versions of a design object may exist in a design database. Thus, in transaction management, version control has to be an integral part of the system.

- The conventional model of atomic transactions are not suitable for modeling the co-operation that exist in design environment, where designers (design processes) co-operate to produce the final result.

- The single request-response interface between the parent and child transactions of a nested transaction model is not suitable for modeling the 'conversation' that may exist between the user (parent) and the application (child) [80]. The 'conversational interface' is explained further in Section 4.

In this paper, we propose transaction management scheme that incorporates version control to maintain database integrity. We propose a nested transaction model that provides for:

1. Conventional parent-child relationship between transactions [60].
2. Transactions with multiple commit points.

3. Co-operating transactions with conversation interface.

4. Subcontractor transactions that execute concurrently.

The rest of the paper is organized as follows. In the next section we present the basic model. In Section 3, we introduce the notion of co-serializable transaction schedules and version consistency. In Section 4, we present the transaction model with multiple commit points. In Section 5, we present co-operating transactions with conversational interface and subcontractor transactions. A summary of commands are presented in Section 6, and Section 7 concludes the paper.
THE MODEL

In this section we give a brief description of our model and some basic assumptions. The discussion on serializability and correctness is based on [54].

Derivation Graph and Design Scheme

In a design environment, a design object may depend on other design objects. For example, an object code of a program depends on the source file. A source file itself may use several 'include' files which may contain data structures and macro definition. The dependency between the design objects can be represented by a directed acyclic graph called a design scheme. Design objects are denoted by nodes and dependencies are denoted by directed arcs. A node $O$ in a design scheme represent the class of objects $O$. In case of revisions, where feedback is used to redefine a data object, cycles may be formed due to the resulting dependency. We do not consider such feedbacks and assume that all design schemes are acyclic. We also assume that several design alternates may exist for the generation of any given design object. Thus, a design object may be generated from different sets of objects by different design processes. In addition, there may be multiple versions for each design object. These assumptions result in a design scheme with multiple alternatives for each design object.

In general, it may not be necessary for designers to know about all possible combinations that may lead to the correct design of an object. A correct sequence of design steps that will take the set of inputs and generate the final set of outputs is sufficient. The dependency graph that shows one such alternative for a set of inputs
and a set of design objective is called a derivation graph. (See Figure 1). Thus, a derivation graph is a subgraph of a design scheme with some of the redundant arcs removed. The nodes are explicitly labeled by the object versions involved in the actual derivation. We associate a derivation graph with each design transaction.

Design Objects

Design objects unlike data encountered in conventional database applications such as payroll, inventory control, and banking, are complex objects [51]. Using molecular abstraction technique proposed in [11], we decompose each data object and their relations into a set of related tables and represent them in a tree-like hierarchy of interface specifications, implementation and instances (Figure 2). The specification part of a data object holds the common data that may be shared by multiple implementations. All implementations (versions) of a data object inherit the parent attributes of the hierarchy tree. Only the version specific details are associated...
with the implementations. Users can instantiate copies of the implementation of the data objects with instance-specific data.

![Diagram of Data Hierarchy of Design Objects]

**Figure 2: Data Hierarchy of Design Objects**

In addition to the complex objects encountered in a design environment, design objects can also be composite—a data object may be composed of other data objects—resulting in a composite object hierarchy. The composite object hierarchy and the interface-implementation-instance hierarchy induce dependencies in addition to the data dependencies specified by the design schemes. Therefore, to maintain consistency of the design objects, database management schemes have to enforce additional control [50]. The next section gives a brief description of the database and the correctness criteria used in this paper.
DATABASE AND CONSISTENCY

The design data that is shared by the designers is maintained by a central database called the *public database*. Design objects in public database are relatively stable objects and can be used by other designers. Design transactions that access the public database *check-out* objects that are to be written/modified and then later *check-in* the objects. The database manager of the public database controls the access to the public database by design transactions. The sequence of operations involved in using the objects in the public database are discussed in [50]. The design transactions that access the public database also maintain their own *private database*.

When a design transaction $T_i$ is invoked, an environment $env_i$ is associated with the transaction. Objects checked out of the public database on behalf of a design transaction are placed in the private database $env_i$. The objects in $env_i$ can be accessed by the transaction $T_i$ and also by the subtransactions of $T_i$, depending upon the relationship between the parent transaction and the child transactions. Associated with $env_i$ is a system-wide unique timestamp $uid_i$. All objects that are created by the design transaction are placed in $env_i$ before they are checked-in into the public database. Operations involving the private database are discussed in subsequent sections.

Serializability

The correctness criteria for conventional database is based on *serializability*. It is assumed that transactions, in the absence of concurrency (and failures), maintain database consistency; the transaction takes the database from one consistent
state to another consistent state. Therefore, serializability criteria requires that only those transaction schedules that are ‘equivalent’ to some serial schedule are accepted. While there are several notions of correctness, view serializability is generally accepted as the correctness criteria for conventional models. However, the entire class of view serializable schedules cannot be recognized efficiently (unless P = NP) [64] and therefore schedulers are implemented for a subset of the class known as conflict serializable schedules.

In a design environment, serializability requirements are too restrictive [78, 54]. Furthermore, the existence of multiple design versions, the long duration of the design processes and the trial-and-error techniques employed make view serializability unattractive. We propose a correctness criteria called co-serializability, where a schedule \( S \) of transactions in set \( T = (t_1, t_2, ..., t_n) \) is co-serializable, if each transaction \( \in T \) behaves as if it is executing atomically. Let \( T = (t_1, t_2, ..., t_6) \). Let each transaction \( t_i \) read and write the same set of data object \( O \). Consider the dependency graph \( G \), where the nodes denote transactions, and arcs denote dependency such that if \( t_i \) reads the data value written by \( t_j \), then we denote the dependency by a directed arc from node \( t_j \) to node \( t_i \). Then if view-serializability is enforced, the dependency graph of a schedule would be a graph representing the total ordering of transactions in \( T \) (Figure 3a), while under co-serializability, the schedule would generate a tree like dependency graph (Figure 3b).
Consistency

In design databases, the design objects are maintained by a central database. Transactions (design applications) access these objects and use them to 'derive' other complex objects. Since complex design efforts are often iterative, other transactions may read data objects and redefine (refine) them, creating alternate versions. Also, transactions using different design principles/techniques may be used concurrently to develop alternate versions of which only one may be used in the final design. Therefore, one can envision a history of versions associated with each object. Since several versions exist for each object, during derivation of a complex object, only consistent versions of objects need to be used. Informally, internal consistency deals with restricting the set of objects read or used during the derivation of an object [52, 54].
In our model we assume that every transaction reads a non-empty set of data objects and writes a non-empty set of data objects. The read-set of a transaction $T_i$, denoted $RS(T_i)$, is the set of objects that are read by the transaction $T_i$; the write-set of transaction $T_i$, denoted $WS(T_i)$, is the set of objects written by the transaction $(T_i)$. We will now recursively define internal consistency of an object and mutual consistency of a pair of objects. Informally, an object is internally consistent if it is in an acceptable state, and has reached that state through a set of acceptable steps, whereas any two objects that are internally consistent are mutually consistent if they are 'compatible'. Note that when there are design alternates for each design object, all internally consistent objects need not be mutually consistent with each other. As an example, consider the implementation of a VLSI circuit. Many different circuits may exist for the same functionality. Thus, the routing information and connectivity details of a design may be mutually consistent with each other if they correspond to the same design alternative, but may not be mutually consistent with those obtained by an alternate (but still correct) design. The notions of consistency are formally defined below.

**Internal Consistency:** If all objects $e \in RS(T_j)$ are pairwise mutually consistent and internally consistent then $O^j_i \in WS(T^j_i)$ is internally consistent. (An object $j$ written by transaction $t_i$ is denoted $O^j_i$.)

Mutual consistency is defined recursively as follows:

1. The initial object $O^0$ is mutually consistent with every object $O^j_i$.
2. Any two versions of an object are not mutually consistent: $O^j_i$ is not mutually consistent with $O^j_k$ if $k \neq j$. 
3. If \( \forall v, \forall u, u, v \in RS(T_j^i) \), \( v \) and \( u \) are internally consistent and mutually consistent then \( WS(T_j^i) \) is internally consistent and mutually consistent with \( u \) and \( v \).

Version control mechanisms enforce such constraints generally encountered in design database applications. None of the existing schemes found in the literature ([12, 26, 47, 56]) have an automatic version control; they require explicit control information from the users. This requires the user to keep track of the different versions of all the data objects that are being used, placing considerable burden on the user.
NESTED DESIGN TRANSACTIONS

To enhance the degree of concurrency in the system, and to model the cooperation that exist in a design environment we propose the following:

- Use nested transaction model.
- Use co-serializability criteria for scheduling the transactions.
- Use multiple versions for design objects.
- Use subcontractor transactions.

Nested Transactions

Nested transactions have been proposed \cite{60, 57, 13, 49} to increase the degree of concurrency and recovery in long transactions. In this scheme, the top level transaction (root or top-action) contains complex operations which can then be decomposed into more primitive operations such as reads and writes. Thus each complex operation in a top-action can be modeled as a subtransaction, which in turn can be decomposed further into other subtransactions, resulting in a tree-like hierarchy.

The tree-like hierarchy of transactions and subtransactions allow designers to interact in a limited way \cite{8}. A designer can request work from another designer by creating a subtransaction (child) for the requested work. The requested work, on completion (of the child subtransaction), becomes available to the requesting designer (parent transaction) and becomes part of the transaction. If the child subtransaction aborts, the parent can create another child subtransaction and wait for the result;
the parent transaction need not abort immediately. Concurrency can be increased by invoking multiple child transactions in parallel. Database consistency is maintained by requiring serializability at every level of the nested hierarchy [13]. If the top-action aborts, the effect of all the child subtransactions are undone. Thus, the commit of the subtransactions are dependent on the commit of the parent. While the 'all-or-nothing' approach may be suitable for conventional data-processing applications, it is often necessary for a transaction to effect permanent change to parts of a database, irrespective of the final outcome of the top-level transaction. A simple example is the need for log-file in a transaction system to record the execution traces of transactions. The need for transactions to have side effects prompted the designers of the ARGUS system [55] to support a nested top-action in which the subtransaction can commit independent of the calling top-action. Hence forth, we would refer to such subtransactions as independent subtransactions. In addition, we identify the following reasons which necessitate a mechanism for supporting independent subtransactions.

- A design transaction may be invoked to design a set of objects. Parts of the design that are complete and independently consistent need to be made available for other designers.

- Design projects that fail eventually could still generate design objects that may be useful (and hence used) in other design efforts.

- Design interfaces that are developed may be made available for subsequent development.

- The effect of a design project may have to be recorded (for further study, accounting etc..) irrespective of the final outcome of the project.
The following types of parent-child relationships are supported by our model of project transaction.

1. Conventional nested transaction, with parent-child relationship as proposed in [60]. The top-action or the root transaction is called the project transaction and its children are referred to as subtransactions. The project transaction can invoke other applications as subtransactions and has to wait for their termination. The children inherit the data objects and the locks from their parents and on completion, return the new data objects and the locks back to their parents.

2. A transaction can invoke a child transaction that can commit independently. In this case, the commit of the child is not subject to the commit of the parent; the subtransaction can commit making permanent changes to the database, while the calling transaction itself may fail.

3. A conversation subtransaction can be invoked by a parent, where the interface between the parent and the child is 'conversational'. In conventional models, the interface between the parent and the child supports a single request (from the parent when a request is made) and a single response (from the child when the results are returned). Between the calls, the data values are stored for recovery purposes. Thus, if the request fails, the data values that existed before the call was made persist; whereas if the request succeeds, the data values are updated by the parent and the new values are saved. In the case of conversation subtransactions, the parent makes a series of requests for the responses from the same child. For example, a transaction may first select a set of data objects, test the data objects (possibly invoking another subtransaction), make changes to
the objects, and then save the objects. In this case, it is assumed that the same subtransaction (application) supports both the select and update processes. Thus it is possible for a parent transaction to have a series of request-response interaction with its child.

4. Subcontractor transactions are subtransactions that are invoked by a parent in which the subcontractor and the parent can be active concurrently. In conventional models, the calling transaction is blocked until the child subtransaction terminates. In [80] a relationship is proposed where the parent and the child can be active concurrently as long as the operations do not conflict with each other. In our model, we support a scheme where irrespective of the operations, a parent can invoke a subcontractor transaction that proceeds along with the parent. In addition, the range of access allowed to the subcontractor transaction can be controlled by the calling parent transaction. Unlike conventional model, the data objects produced by the subcontractor transactions are not automatically inherited by the calling parent. This model is useful for scheduling alternate design operations concurrently on the same data objects.

In the subsequent sections we describe each of the models in detail and show their correctness under the framework of the proposed model.

**Multiple Commit Points**

A transaction can invoke a child transaction that can commit independently by assigning the child transaction its own 'commit sphere'. Informally, a commit sphere \( C_S_i \) is defined as a set of transactions such that if \( \exists t_i \in C_S_i \), if \( t_i \) commits, then \( \forall t_j \)
\( \in CS_i, t_j \) will also commit. In conventional models, the commit sphere contains the set of transactions that are involved in the 2-phase commit protocol. When a child transaction is assigned its own commit sphere, it does not inherit any of the parents\ data objects from the parents commit sphere. The child transaction now has to compete with other transactions in the system to acquire locks on the objects that it may access. The parent process waits for the termination of the child. However, the locks and the data objects acquired by the child process are not returned to the parent after the child process terminates. The child returns the status of the termination (success or abort) to the parent. In the case of a failure of the child, the parent can attempt to invoke the call again; in case of success of the child, the parent can use the results that are placed in the public database by requesting locks on the data objects.

Consider the design graph DG for data objects \{a,b,c,d\} with the dependencies shown. The objective of the project transaction is to generate a set of mutually consistent design objects \{a,b,c,d\}. The design effort can produce objects a, b and c successfully and may fail to produce object d. The object d can be produced by another project transaction during subsequent effort. Therefore, it is necessary to save the design objects a, b and c independent of d (if necessary). This can be achieved if the objects a, b and c are designed by child subtransactions that can commit independently.

**Correctness**

While the correctness of a transaction with independent commit spheres is easily shown in the case of models proposed in [57] or [80], it is not the case in our model.
In conventional models, the correctness is maintained by enforcing serializability, and requiring the child transaction to behave like a top level transaction. However, in our scheme we have multiple versions of the design objects and co-serializability is the correctness criteria. Therefore, if the project transaction PT\(_i\) was invoked on a database D, then, irrespective of the outcome of the top level transaction PT\(^{\text{top}}\), the database has to maintain consistency, even in the presence of independent commit subtransactions. Thus, the set of objects that are written by the child transactions that commit independently has to be internally consistent and mutually consistent. This can be enforced by the following protocol given below.

\textbf{Protocol Independent-Commit} Let ICS\(_i\) = \{it\(_{i1}^u\),...,it\(_{ij}^z\}\} be the set of subtransactions that can commit independently. Let ICWS\(_i\) = \{o | \exists it\(_{ik}^o\) \in ICS\(_i\) and it\(_{ik}^o\) has committed\}. Then we require \(\forall u, \forall v, u \in ICWS\(_i\) \Rightarrow (u \text{ and } v \text{ are mutually consistent})\).

That is the set of subtransactions which is a subset of ICS\(_i\), should commit in the order specified by the design graph, and the subtransactions should read version consistent objects from the public database.

When an it\(_{ik}^o \in ICS\(_i\)\) attempts to commit, then its write-set is verified against the ICWS\(_i\), the set of objects written by transactions that have already committed. The objects that are written by the ICS\(_i\) should be made available to the project transaction PT\(_i\). To maintain mutual consistency of the objects read and written by other parts of the project transaction PT\(_i\), when a read is invoked by a subtransaction for a data object \(o \in ICWS\(_i\), it is required that they read the versions written
by transactions in $ICS_i$. In addition, if $PT_i$ has to commit, it is necessary for parts of $PT_i$ not in $ICS_i$ to write objects that are version consistent with the subtransactions that have already committed. Therefore, based on the definition of mutual consistency, we require that objects written by the transactions $PT_i$ to be mutually consistent with $ICWS_i$. That is, $WS(PT_i) \cap ICWS_i = \emptyset$.

While this requirement may appear to be too rigid, it is necessary. Let $O \in ICWS_i$ and $WS(PT_i)$. There are two cases to consider: (1) $O$ is written by a subtransaction in $PT_i$ before a subtransaction $it_i^O \in ICS_i$ attempts to write another version of it, and (2) a subtransaction of $PT_i$ attempts to write $O$ after it has been written by $it_i^O \in ICS_i$. In case (1), the conventional model would result in a deadlock as the transaction $PT_i$ would not have released the $W$ lock on the object and $it_i^O$ would wait for the $W$ lock to be released by $PT_i$. Therefore, the transaction has to be aborted without generating the object $O$. In case (2), the conventional model can request a $W$ lock on object $O$ and may attempt to write a new version of the object. However, in our model we assume that a transaction generates only one set of consistent objects. If the object which has already been released to the public database by $it_i^O$ is rewritten by same project transaction $PT_i$, in effect two versions of the same object are generated resulting in version inconsistencies. Therefore, it is disallowed.
COOPERATING AND SUBCONTRACTOR TRANSACTIONS

In conventional nested transactional model, the subtransactions are atomic with respect to their siblings. That is, if \( t_{i1} \) and \( t_{i2} \) are the child transactions of a sub-transaction \( t_i \), then \( t_{i1} \) and \( t_{i2} \) are atomic with respect to each other- \( t_{i1} \) precedes or succeeds \( t_{i2} \) in the execution history of \( t_i \). However, in design environment, a top level transaction may require partial results from one of its subtransactions which may then be used to determine the course of subsequent actions. The partial results may be passed on to other subtransactions or the result may form the basis of the next request to the subtransaction. In other words, the subtransactions cooperate by conversing with each other.

Another scenario involves a subtransaction that may request multiple subtransactions to perform similar tasks. A transaction (client) may request several subtransactions (subcontractors) to perform a task and eventually select only one of the outputs of the subtransaction. The set of outputs that are finally selected by the client may be determined by some selection constraint(s): the first to complete or the least expensive or the one that satisfies the most constraints. In conventional model, this involves sequential execution of the child subtransactions or a renaming convention to avoid overwriting or lock conflicts. In addition, the access of the subtransactions are not controlled by the parent subtransaction. However, in design environment subcontractors are often requested to perform certain tasks, independent of the siblings, with restrictions on the access. This is modeled by the subcontractor transaction model. The two models are discussed next.
Conversation Interface

In conventional applications, a transaction (we shall refer to the invoking transaction as the user transaction) invokes another subtransaction (we shall refer to it as the application) with a request. The request is carried out by the application and the result is returned. The states are saved by the user transaction before the call is made and after the call is complete. In case the call fails and the application returns a failure, the state information saved is used for restoration. This model is called the single-request model. The intermediate states generated during the execution of the request are not saved by the user transaction. Thus, in the case of failure of the application, the user transaction is unaffected by the failure and the call can be retried. Also, the effect of the application is perceived by the user transaction to be atomic. However, in design databases, it may be necessary for applications to communicate with the user transaction. There may be multiple request-response sequences between the user transaction and the application transactions. The user transaction may be able to observe intermediate effects of the application.

Consider the following example. A user transaction can invoke an application to do the following:

User-request: selects all flip-flop gates

Application-reply: (gate-1,...,gate-n)

User-request: update gate-i....

Application-reply: Yes

...
In such cases, the intermediate states generated are visible after each response. During the request-response sequences, the intermediate states generated are not saved. The results returned by the application transactions can now be used by the user transaction either locally or passed to another application (invoked by the user). This allows for sharing of design objects that are not complete or consistent in some sense and hence may not be made available globally. In the example discussed above, the gate-i, which may not be complete now (due to the updates made) may be made available to another application, such as a design-rule-checker, to test for correctness. Or alternatively, the user application could, by using another application design a modified version of gate-i and replace the old version with the new version of gate-i.

**Recovery and Backout Sphere**

In case of failure of a transaction, the system state has to be restored to maintain the database consistency. In nested transaction models proposed earlier (e.g., [60, 67]), in case of failure of a subtransaction, the subtree rooted at that subtransaction has to be aborted and the system has to be restored to the state that existed before the invocation of the subtransaction. We shall now define a term *backout sphere* which was introduced in [80].

A backout sphere $bs$ is the smallest set of connected transactions belonging to the transaction hierarchy such that if any member $t_i \in bs$ fails, then all transactions $\in bs$ have to be aborted to restore consistency. Note that in conventional models, if a parent transaction fails, then the child transaction also fails even if the child does
If the parent-child interface is a single request-response type, as in the case of conventional nested transactions, the states are saved before and after the requests, and the subtransaction itself is atomic with respect to the invoking user transaction. Let $t_i$ be a user transaction and $t_{ij}$ be a subtransaction of $t_i$. If the subtransaction $t_{ij}$ fails, the state information saved before the invocation is used to recover and the request may be retried. The parent transaction $t_i$ itself is unaffected. Therefore, the backout spheres in such a hierarchy are singleton sets of the subtransactions. Figure 4 shows the backout spheres of a nested transaction.

In the case of conversational interface between the parent and the child, due to exchange of data values between the child subtransactions and the parent transac-

![Figure 4: Backout Spheres](image-url)
tions, the intermediate states between the exchanges are not saved. In case of a failure of a child, the parent transaction has to abort as well because the state before the failure cannot be recreated by the parent or the failed child transaction alone. This is similar to the cascaded rollback scenario encountered in communicating processes discussed in [66] where a failure of a process may result in rollback of all processes that were communicating with each other.

Due to the cost involved in the saving of all intermediate states generated during the execution of cooperating transactions and the processing involved in computing valid recovery lines, the solution proposed in [66] is not applicable in our model. Furthermore, failure of a request to a subtransaction $t_i$ might require abortion of the entire transaction whose results might be used by other subtransactions that are still active. In this case, aborting transaction $t_i$ requires aborting all other transactions that would have accessed the partial results generated by $t_i$. Thus the set of connected transactions in the nested hierarchy with conversational interface should all belong to the same backout sphere. Also, unlike the model proposed in [80], where a subtransaction $t_i$ with conversation interface could commit independent of the calling transaction $T$, we do not allow the child transactions with conversation interface to commit independently.

Implementation

Let $T$ be a transaction which is the root of the subtree that has conversational interface with its children. Let $t_i$ be a subtransaction of $T$ and $t_{ij}$ a subtransaction
Let \( \langle x; z \rangle \) be the sequence of request-responses observed at the interface of \( T \). Then we assume that under total sharing of data objects, the system will maintain consistency if each request is executed atomically while the state information is not retained between the requests.

Let \( x_1, \ldots, x_i \) be the sequence of request-responses generated due to request \( x \) by some transaction \( t_j \) to which the request \( x \) was directed. Let the maximum nesting of transactions with conversational interface be \( n \) such that request \( x_{1,2,\ldots,n} \) is a single request. Then to maintain consistency, we require that the requests in the hierarchy are serializable in the order generated by the lexicographic ordering of the strings using the ordering : \( (x_{1,1}, \ldots, x_{1,2}, \ldots) \) This can be ensured either by attaching a timestamp with each request to the subtransaction such that the access to the data objects are serializable in the order of the timestamps or by using locking. In case of locking, we require that the subtransactions use two-phase locking during each interaction. (i.e. between each request-response). Furthermore, local variables have to be discarded between the calls. It can be shown that if the set of transactions belonging to a nested tree with conversational interface follow the protocol discussed above, they maintain correctness.

When a parent transaction \( T \) decides to invoke a child transaction \( t_1 \) with a conversational interface, it passes to the child transaction its environment that has been created when the transaction \( T \) was created. If \( T \) itself had been called by another transaction \( TT \) with a conversational interface, then it would have inherited the environment of \( TT \) which is passed to \( t_1 \). When transaction \( T \) makes a request
(a;u-id) to t₁, the set of objects accessed are restricted to be compatible using the protocol discussed earlier. The data objects accessed are placed in the environment along with the locks with the unique identifier (u-id), associated with the call. The transaction completes the call and when the call is returned, the objects acquired and the locks are left in the environment. The calls are synchronous in that the calling transaction is blocked and waits for the call to be complete. Once the call a is complete, the set of objects that are accessed are left in the environment, and the parent inherits the objects and the locks obtained due to the execution of the call a.

The transaction T can now use the objects (and the locks) generated by t₁ and can issue another call b to t₁. Between the calls a and b, the application does not retain any state; the state information that may be used and saved during the execution of call a are lost, and may have to be regenerated from the environment. Thus, the application should not have any local variables that may be saved between calls or should use functions to compute state values.

An Example: Let T be a design transaction that uses two other subtransactions t₁ and t₂, both with conversational interfaces. Let t₁ be a transaction that is used to design an interface for a VLSI gate and t₂ be a transaction that is used to interactively test the design rules of a gate. Let T invoke a on t₁ where a results in the generation of an interface gi. After execution of a, the design is returned to the calling transaction T which then invokes c on t₂ with gi as a parameter. The subtransaction t₂ may then test interface gi for correctness and may report the validity of the design to T. The transaction T may then choose to correct/improve on the
design of $gi$ and then return the object $gi$ to $t_2$ to further improve or complete the design by invoking a call $b$. During the execution of the call $b$ in transaction $t_2$, all local variables including the data object have to be reevaluated; those that are used during the execution of the call $a$ are lost once the call is returned.

Now we introduce the following algorithm for generating backout spheres in a nested transaction model for generating backout spheres.

Algorithm:

Notation: Let the top level transaction be denoted as $t_1$. Denote the children of $t_1$ as $t_{11}, t_{12}, ..., t_{1n}$. The $nth$ child of sub-transaction $t_{ij}$ is denoted as $t_{ijn}$.

Input: $TS$, a set of subtransactions of $t_1$.

Output: Sets of transactions grouped into backout spheres.

begin
index := 1;
$bs_1 := \{t_1\}$;
while ($TS \neq \emptyset$) begin
begin
$\forall t_i \in bs_{index}$
$\forall t_{ij} \in TS$, if $t_i$ is the parent of $t_{ij}$ and $t_i$ has conversation interface with $t_{ij}$

begin
$bs_{index} := bs_{index} \cup \{t_{ij}\}$
$TS := TS \cdot \{t_{ij}\}$
end
end
end

end
index := index + 1;
Select t_i from TS such that i is the smallest subscript;
bs_{index} := \{t_i\}; TS := TS \cdot \{t_i\};
end
end

Note that if ts_i is the set of subtransactions of a nested transaction T_i, then there exists a unique maximal partition of ts_i into backout spheres. This follows from the construction algorithm presented above.

**Subcontractor Transactions**

In a design environment, a user may require the help of other entities that may act as independent entities. Often, a task may be assigned to a subcontractor who may have restricted access to the data objects. Several subcontractors may be assigned the same task and only some of the results may be eventually accepted by the client. In this case, all subcontractors may be concurrently active, generating similar data objects. This can be modeled by a parent-child relationship called client-subcontractor model. In this model, the parent transaction (client) invokes subtransactions t_i (subcontractor) with a request (req), an environment variable env_i, and a set of constraints C_i called the commit constraints. The environment env_i provides the subcontractor with a set of capabilities to access data objects. In addition, the set of starting values that may be used by the subcontractor transaction is also placed in env_i. The subcontractor is concurrently active with the calling client. Thus several subcontractors can be active at the same time. The locks obtained by the
subcontractor transactions are local and do not conflict with the locks of the parent (client) or other subcontractor transactions.

Thus, in this model, it is possible for a set of processes to access a set of design objects and hold locks in conflicting mode. When the subcontractor terminates, it returns the env to the calling client which can then accept the result or reject the result. If the results in an env\(i\) are accepted (by verifying against the commit constraint \(C_i\)), then the objects generated by the subcontractor are placed in the parents environment; else they are discarded. The minimum requirement of the commit constraint \(C_i\) is the internal consistency requirements for objects in env\(i\) and mutual consistency of objects in the parents env and env\(i\).

If the client transaction terminates (aborts or commits) before the subcontractor completes its call, the results of the subcontractor are discarded. The subcontractor transaction \(t_i\) can in turn have other subtransactions with one of the three parent-child relationships discussed earlier (except independent commit); the environment of the child transactions will now be based on env\(i\).
OPERATIONS

In this section we give a brief summary of the operations that are allowed within a design transaction. We assume that a unique identifier is associated with every transaction. The identifier is a function of time and can be used for implementing static ordering of accesses to data items as proposed in [67]. Also, associated with every top-level design transaction is a design graph \( dg \), which denoted the data dependency constraints that are explicitly denoted and hence verified by the system. A process can begin a top-level transaction by using the command: \texttt{begin-transaction}(uid, dg), where uid is the identifier associated with the process/transaction and dg is the design graph. When a transaction is initiated, an environment (private database) \( env \) is associated with it. The objects that are checked-out of the public database by the transaction are placed in the \( env \). When the transaction commits, the objects in \( env \) are checked-in, subject to the correctness criteria enforced by the public database. A top-level transaction (root) can create subtransactions by the call \( id = create-child(env, request) \). The call returns an \( id \), the identifier for the subtransaction. The call takes two parameters: (1) an environment, and (2) the name of the request along with the parameters. By varying the environment, the different parent-child relationships discussed earlier are possible. They are now presented below.

1. \texttt{create-child}(\texttt{NEW}, \texttt{REQ(params)}): The keyword 'NEW' assigns a new environment but with the same design graph as the calling transaction. The new environment, inherits all the attributes from the calling environment. When the child transaction commits, the call returns with the non-null value for the
id, and the child transactions environment is accessible through the return id.
The calling transaction can now access the objects in the child transactions
environment and if the design is acceptable, add the result to its environment
by incorporating the design by the call inc(id). The call inc(id) updates the
local environment with the objects from the environment associated with the
identifier id. The child transaction created in this mode behaves similar to the
child transaction suggested in [60].

2. create-child(OWN, REQ(params)): A child transaction created in this mode has
its own commit sphere and can therefore commit independently. It does not
inherit any objects from its parent transaction. A new environment is created
with the same design graph as the parent and the request 'REQ' is executed
in the environment. When the child commits successfully, the objects in its
environment are checked-in into the public database.

3. create-child(SELF, REQ(param)): A child transaction with conversational inter­
face can be invoked by this call. Application (subtransaction) named REQ is
invoked with 'params' as parameters. The call returns immediately after the
application is initiated with the 'id' that identifies the application. Subsequent
requests to the application can be made through: 'id.request-name(params)'.
The call is executed and the child inherits the environment of the parent and
makes permanent changes to it. Note that in this scheme, the child does not
have an explicit environment (private database) and uses the parents environ­
ment. Therefore, no explicit inc() operation is necessary at the end of the
call.
4. \texttt{create-child(env_i,REQ(params))}: A subcontractor child transaction can be invoked by creating an explicit environment \textit{env}_i and then assigning it to the application 'REQ'. An environment can be created by the call \texttt{new-env(capability, dg)} which returns an environment \textit{env}_i. The 'capability' is the access privilege that is assigned to the environment; the access of the subcontractor can be controlled by suitable specification. The design graph \textit{dg} can be a sub-graph of the parents design graph. The parent can also place starting values of data items in the environment created so that the values can be used by the subcontractor transaction. The subcontractor transactions run concurrently with the calling client transaction. The client transaction can wait for a subcontractor child by a \texttt{wait(id)} call. Once the subcontractor terminates successfully, the results are available for the parent to incorporate into its database. The parent transaction, can however commit without waiting for the subcontractor child transaction. In that case, the environment associated with the subcontractor is discarded. A subcontractor transaction cannot invoke a subtransaction that can commit independently.
CONCLUSION

In this paper we propose a model for design transactions that extend the conventional parent-child relationship in nested transaction models proposed earlier. The following concepts are proposed. First, multiple commit points allow for side effects and reduce the cost of transaction failures. Second, conversation between design transactions allow cooperation between design processes. Third, a subcontractor transactions that capture effect of concurrent design alternates. A correctness criteria and control mechanism for implementing the different models are also presented. These models are appropriate and necessary for design transactions, and are more general than those that have been proposed earlier.
SUMMARY AND CONCLUSION

In this dissertation, we first identify the issues that are encountered in applying the conventional database and transaction model to design databases. We maintain that the problems are mainly due to (1) the nature of the complex data encountered in design applications, and (2) the long duration design transactions. In addition, the nature of the design environment itself poses some problems.

We present two models of design transactions: uniform transaction model and non-uniform transaction model. We show that existing version/revision control mechanisms such as make in the Unix system, do not adequately support the scenario we present under the two models. In addition to the conventional notion of consistency based on serializability additional controls are needed.

We formally present the notion of version compatibility. Acyclic design schemes with a unique source (base object) and a unique sink (final object) can be modeled by the uniform transaction model. In this model, write-sets of transactions are unique. As an example, the set of VLSI CAD tools can be modeled as a uniform transaction model. When several design alternatives are available for each phase of the design, resulting in set of versions for each object type, coherent schedules maintain version
consistency. This can be used as the correctness criterion in situations discussed, in the place of *view serializability* which has been used in conventional databases.

In the non-uniform model, acyclic design schemes can have multiple sources and sinks; the write-sets of the different transactions are non-empty. In this scheme, we show that determination of the set of transactions, and hence the schedule to derive non-redundant set of derived objects is NP-hard. We also show how internal consistency and mutual consistency of design objects can be used as correctness criteria for maintaining consistency of multiple versions of design objects. We also present a mechanism for maintaining on-line version control which can be used to generate correct schedules in the presence of concurrent design transactions. The proposed version control scheme is modular and can be implemented efficiently, independent of the concurrency control scheme. However, we show the limitation of the model when data objects are not stratified into several levels of abstraction. A change applied to a data object results in a creation of a new version of the object, and this may necessitate redesign of all other objects that depend on the original data object. Thus additional modeling concepts are necessary to model design data.

Modeling concepts for complex objects, such as representation schemes for storing data associated with design objects are presented in Part 3. We use examples from VLSI and electronic circuits to illustrate the concepts. By stratifying design data into a hierarchy of interfaces and implementation, we allow for concurrent access to different parts of the data, while avoiding representation redundancy and unnecessary change propagation or redesigning of objects. We propose the concepts
of parametric versions that facilitates the reusability of a set of design objects - the parametric versions/interfaces to obtain several versions or instances of the object type but with significantly different characteristics. We also propose the concept of conditional instantiation which can be used to maintain integrity even in the presence of changes in object versions.

A scheme for managing long-running design transactions (project transactions) on the public database is presented in Part 4. Furthermore, hypothetical transactions - a design transaction that generally aborts - is supported by the scheme. Protocol for upgrading the hypothetical transaction to a regular transaction is also provided.

Composite objects are grouped into objects called Atomic Subtrees (AST). Elements of an AST are inter-related and therefore, to maintain consistency of the composite objects, operations on the elements of the set have to be atomic. A protocol for accessing and manipulating composite objects is also presented.

The different models presented for controlling a design transaction into a tree-like hierarchy of subtransactions with possibly multiple commit points, and different recovery spheres allow for structuring design processes to reduce rollback. Furthermore, the conversational interface allows for interaction between subtransactions. The models presented encompass the conventional parent-child relationship found in [67, 60].
FUTURE WORK

The model presented here can be supplemented by providing the following. In addition to the set of primitive operations suggested for data manipulation, a query language capable of supporting complex operations need to be developed to support the model.

Suitability of other data models could be explored for adapting it to support complex objects. Object-oriented approaches, as already pointed out, are becoming increasingly popular. Once a formal framework and notions are established, they could easily supplant the data model used here. One of the serious drawbacks of the E-R model used here, for storing complex data, is the large number of tables (relations) and the joins that need to be performed to extract the complex attributes. Some extensions to the E-R model along the lines of $\text{NF}^2$ approaches may be provided to alleviate the problem.

The third issue that could be addressed is the change-propagation in case of design changes. We show that due to the abstraction hierarchy, the need for change-propagation is considerably reduced. In addition, parametric and conditional instantiations also obviate the need for propagating change in most cases. However, there are still situations in which the dependent objects have to be redesigned. While several alternatives have been suggested in literature, we do not attempt to discuss them here.

An important consideration that has not been addressed so far in this work is
data security. The public database that is the repository of design data is shared by all 'designers'. In addition, the data in the database has considerable commercial value. Thus security of the data from intensional and unintensional corruption or misuse has to be prevented. This is a significant problem when our client-subcontractor model supports even users outside the realm of the local network. While a security model based on object identifier is implicitly available due to the consistency control mechanism, it is evident that a security mechanism has to be developed for the proposed model. A mechanism for controlling the interaction between design transactions have been suggested in [24], where access control forms the basis of cooperation between designers. As object-oriented systems become wide-spread, issues related to data security as applied to object-oriented paradigm will be addressed and solved [75]. Future work in this area will address issues discussed above.
BIBLIOGRAPHY


