TRANSACTION EXECUTION DEPENDENCIES IN A MULTIDATABASE ENVIRONMENT

BY

PRASANNA GOVINDANKUTTY

A DISSERTATION
SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE
THE UNIVERSITY OF MANITOBA
WINNIPEG, MANITOBA, CANADA
MARCH 2001

THE UNIVERSITY OF MANITOBA

FACULTY OF GRADUATE STUDIES

*****

COPYRIGHT PERMISSION PAGE

Transaction Execution Dependencies in a Multidatabase Environment

BY

Prasanna Govindankutty

A Thesis/Practicum submitted to the Faculty of Graduate Studies of The University of Manitoba in partial fulfillment of the requirements of the degree of

Master of Science

To Kamal with love...

ABSTRACT

Advanced transaction models have been the focus of research in the area of transaction management. Examples of advanced transaction models include nested, multilevel, flex transactions, *etc.* However, these models have their own shortcomings that can be summarized as follows:

- Absence of a mechanism to specify and provide proper global integrity constraints (dependencies) that determines the effects on global atomicity.
- Unsuitability of certain transaction models in multidatabase environments. For example, ConTract model was developed for cooperative environments and are thus not suitable for multidatabase environments.
- A means to characterize the amount of local autonomy affected. For instance, Sagas do not address characterization of local autonomy.
- A mechanism to utilize the application semantics and execution dependencies. For example, nested transactions do not explicitly address the use of semantics.
- A provision to support multiple transaction execution alternatives. Sagas and ConTract are examples of those models that do not provide support for functionally equivalent transactions.
- Scalability to Internet environments running advanced database applications. Most of the transaction models presented in the literature do not address the scalability issues.

This thesis presents a novel, and Internet-scalable implementation of a nested transaction model that describes the pragmatic components required to remedy the above shortcomings in the form of an abstract model. It shows that utilizing application semantics and revealing partial results in an open nested, multidatabase transaction environment aids in characterizing the dependencies among the child transactions. An interface mechanism to characterize the level of autonomy at the underlying systems is provided. Support for multiple transaction execution alternatives is also provided.

ACKNOWLEDGMENTS

It was by chance that I walked into Professor Ken Barker's office the first time. A fine resource for research ideas and a great source of help in lot of matters, Ken is the coolest supervisor I could have wished for. I thank him very much for all his support in different kinds of matters during the course of my graduate program.

I thank my examining committee, comprising Professor Sylvanus Ehikioya from Computer Science, and Professor Bob McLeod from Electrical Engineering. I very much appreciate their insights into the material presented in this dissertation.

Thanks are due to Jose at TR*Labs*, Winnipeg for his help and support even after I moved to Calgary during the course of my program. The TR*Labs* Scholarship supported most part of the research presented here. Thanks very much to TR*Labs* as an organization for providing the much-needed money when I was living the life of a grad student – the poorest animal on the face of the planet!

I thank everyone at the offices of the Departments of Computer Science at the U of M, and the U of C for all the admin support they extended during this program – especially for sending my pay slips to Calgary promptly!

Being a lazy bag that I am, I had to be motivated time and again. A great deal of gratitude goes to Papps, Chechi, Uma aunty, Prabhu, and Punita for their constant "prodding" so I finish my Masters soon.

Most implementation ideas for this thesis stemmed from my discussions with Juraj, either over coffee at Second Cup in Kensington, or over pints of beer at the Grad Lounge. Though we were slackers when it came to running, we were pretty serious in all these discussions. I owe you one buddy!

Thanks to Wendy for her company when it came to beers at Calgary pubs, Whoppers at Burger King, and occasional movie nights. Of course, this came with some deep discussions on feminism that I honestly enjoyed!

I'm grateful to Kish and Jiten for their friendship while in Winnipeg, and otherwise. A big "thank you" to Binny for being a good friend and supporting me in many different ways.

This section would be incomplete if I do not thank my family. I owe bigtime to my brothers, mom and sisters-in-law for all the support, patience, and perseverance they have shown from across the seas. I must also thank my young nephew and nieces for their sweet conversations over phone. It meant a world of difference during stressful times. You all have enriched my life!

A final word – If I did forget to mention anybody's name on this, its probably because I wrote this after a couple of drinks! It is not intentional.

TABLE OF CONTENTS

LIST OF FIGURES

"Begin at the beginning", the king said, gravely, "and go on till you come to the end; then stop."

- Lewis Carroll, Alice in Wonderland

Chapter 1
Introduction and Preview

1.1 Motivation

The later part of the 20th century observed a remarkable progress in technology pertaining to information access and its use. Specifically, we observed the progress of the Internet architecture and its use for commerce, and the extensive use of databases to serve the commerce itself. The main reason for the progress in the expansion of Internet is the sudden and vibrant explosion of its World Wide Web (WWW) facet. The reason for the extensive use of databases is from the demands for repositories to store the huge amounts of different types of data used by electronic commerce.

It is natural to believe that the combination of these emerging technologies would yield a powerful means for information access. However, at this time, this is not the case. Why? Could it be because Internet expansion is not as functional as we believe it to be? Could it be because the databases that already exist are unsuitable as the vertebra that forms the backbone for electronic commerce? Or is there another reason that the Internet

infrastructure is failing to reach its full potential? The expansion of the Internet is undeniable. Unfortunately, it is limited by the inertia in the advancement of the infrastructure that forms the backbone of the WWW. Most organizations that have the potential to go online are unable to do so because of their own legacy systems. However, the primary reason this combination is not powerful is because of the lack of proper management of electronic transactions that execute in this environment. Since this is the first generation of electronic commerce, many of its participants are still in the reengineering phase from the previous generation of non-electronic commerce. But once this is complete, increased pressures will be placed on the Internet infrastructure.

The problem of transaction management in a multidatabase environment has been around for a long time. This mission critical problem has been addressed by several researchers working on both the concurrency control and reliability aspects, and the development of extended transaction models (ETM). One possible scenario where transaction management is required is an *online information kiosk.* These are interactive web pages that guide the online shopper towards a collection of online stores carrying item(s) of interest. Examples of such a collection of online stores are air ticket bookings or car rentals. A traveler planning a tour could use the information provided by the kiosk. Other situations where transaction management is required are online auctioning, e-business web sites handling major financial transactions, *etc.*

There are many questions that arise when the transaction management problem in MDBS is extended to address transaction management issues in an Internet environment such as an information kiosk or e-business environments. The questions to be answered include:

- How to realize a multidatabase environment on WWW?
- What is the suitable advanced transaction model that can be used?
- How to enhance the parallelism of the transactions in such an environment?
- How to enhance the performance of transaction management systems that are used in Internet business applications?

In this thesis an online tour booking application is used as an exemplar while addressing the above questions. Clearly this application requires a distributed solution. Hence all the information is stored in individual databases and can be accessed by online clients. We use an extended transaction model for executing the transactions in this environment. There are two models of particular interest to this work – Multilevel Transaction Model [WS84][W86][W91] and Open Nested Transaction Model [WS92]. Both these are broadly defined as nested transaction models. The former addresses transaction management issues in multi-layered systems while the latter addresses the problem of enhancing the parallelism of concurrent transactions. Transactions in the MDBS environment (considered in this thesis) are operations invoked by other transactions. Hence, the choice of a nested transaction model will serve our needs well. We must identify the dependencies between transactions (subtransactions) that exist in the nested transaction. The underlying theory in identifying these dependencies is that the transactions (subtransactions) can view the partial results of other transactions (subtransactions) based on certain conditions specified by the application requirements. Further, this choice of model will affect the concurrency control and recovery mechanisms that exist in traditional transaction management systems. This work contributes a (major) step in providing access to information in a multidatabase environment through WWW.

In Section 1.2 we describe the issues involved in managing transactions in a multidatabase environment. Section 1.3 discusses the fundamental research issues of this thesis. Section 1.4 presents an overview of the contributions and describes the organization of the rest of this thesis.

1.2 Transaction Management Issues In Multidatabases

A *multidatabase system* (MDBS) is an interconnection of several autonomous element databases each with its own database management system (DBMS). A *multidatabase management system* (MDBMS) is a software facility developed on top of these element database management systems to provide the users access to any underlying element database. The transaction management problem in MDBS is critical due to the

autonomous and heterogeneous nature of the preexisting legacy systems and their need to support the ACID properties of transactions. Transactions in a MDBS are either local or global depending on how and where they execute (see Figure 1.1).

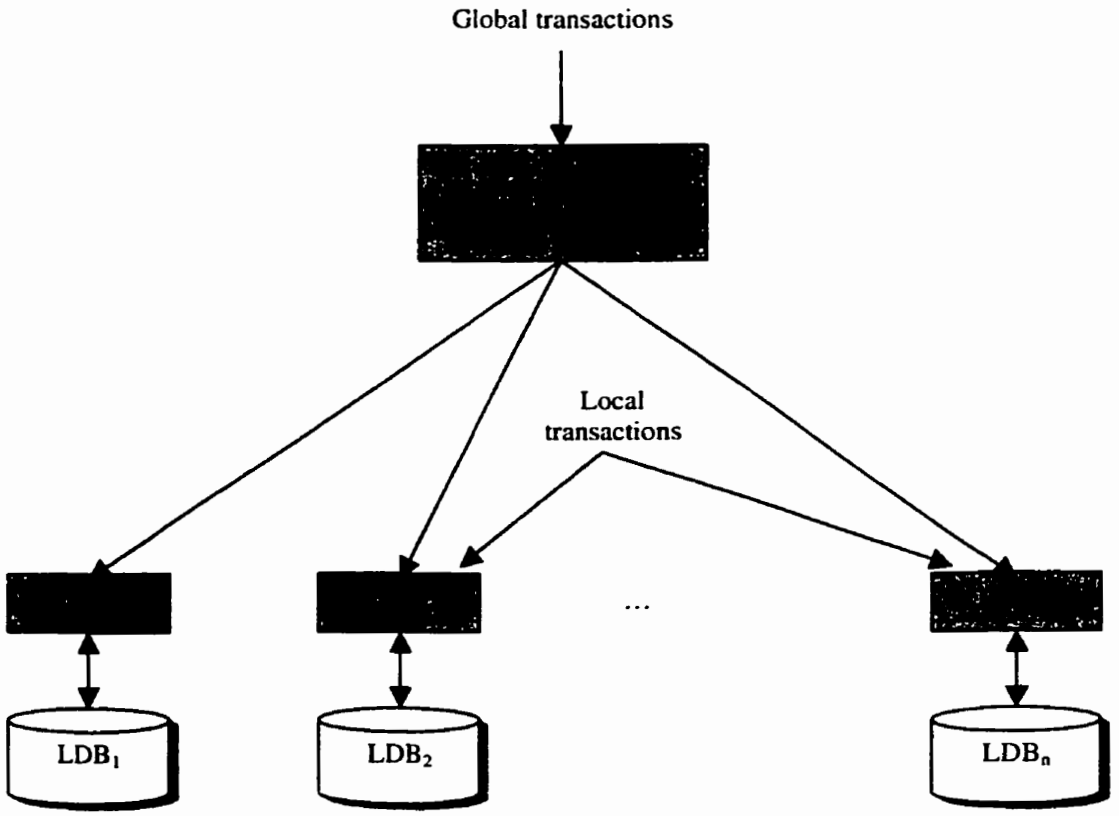

Figure 1.1 A Multidatabase Architecture

Due to the autonomous nature of the underlying systems in the MDBS, the local and global transactions interfere with each other thereby producing undesirable situations or inconsistent database states [BHG87]. A solution to coordinate the execution of global transactions was not made available because the transactions in the MDBS see inconsistent data during their execution. The following sections present the characteristics of the MDB environment, properties of transactions, and the necessary and sufficient conditions for proper transaction management in a MDBS.

1.2.1 Autonomy and Heterogeneity

An MDBS, as defined earlier, is an interconnection of multiple preexisting element database systems. These database systems are autonomous and heterogeneous due to their design, development and administration [BBE99].

Autonomy of the element database indicates the degree to which the DBMS can operate independently without losing control over local data and transactions. Design, communication, execution and association autonomies are the different aspects of autonomy [SL90]. Due to this nature of the element databases (and hence their DBMS), the MDBMS has no control over the following:

- Design of the element databases,
- Local execution schedules,
- Communication between the element databases, and
- Level to which certain functions/operations can be shared with the users of the element databases.

Heterogeneity is another characteristic of the transaction management problem in a MDBS. It refers to the different data definitions, data models, access languages and storage structures that each element database can have. The more dissimilar the two systems are, the more difficult it is to manage that heterogeneity [BBE99].

1.2.2 Properties of transactions

A transaction is a sequence of read and write operations on a database. Transactions have been characterized with the following properties [GR93]:

Atomicity: This property indicates that a transaction either executes to its completion or does not execute at all. That is, a normally terminating transaction makes permanent changes to the database. Otherwise, no changes are made permanent in the database.

Consistency: This refers to the correctness of a transaction. A correct transaction is a program that moves the database from one consistent state to another.

Isolation: This property requires each transaction to see a consistent database at all times. An executing transaction cannot reveal its results to other concurrent transactions before it commits.

Durability: Durability ensures that once a transaction commits, its results are permanent and cannot be lost from the database.

The atomicity and isolation properties support serializability of transaction management, while the consistency and durability properties ensure reliability. Traditionally, it is believed that correctness of transactions can be guaranteed only if all these properties are supported.

1.2.3 Necessary and Sufficient Conditions

The necessary and sufficient conditions for proper transaction management in a MDBS are [B90]:

- All the local database management systems guarantee local synchronization atomicity.
- If an operation of transaction T_1 occurs before T_2 in a DBMS, then the same is true for all other operations whether they conflict or not.
- The global transactions cannot be split and concurrently submitted to the same DBMS.
- The MDBMS must be able to identify all objects referenced by all global transactions.
- The MDBMS must be able to detect and recover from global deadlock.

1.3 Preview: Fundamental Research Issues

The challenges to transaction management in MDBSs are primarily due to the nature of the MDBS architecture and the properties of the transactions. Both these factors raise a number of issues. The rest of this section presents the key thesis element and an overview of the key research issues that affect transaction management in a MDBS. Finally, an outline of the proposed solution to these issues is provided.

1.3.1 Thesis of the Thesis

This thesis broadly addresses the *transaction management problem* in *multidatabases*. Specifically, an *open nested, multilevel transaction model* is applied to a multidatabase environment characterized by the autonomy of the underlying databases. A *set of dependencies* existing between the transactions (subtransactions) in a nested structure of transactions executing in an application-specific domain is identified. Concurrency control aspects of the transactions executing within such a framework are studied. Specifically, the *intra-transaction parallelism* of a transaction executing in such a framework is studied. Further, it shows that by *exploiting the dependencies* identified, a transaction can be processed in many different ways. Finally, an interface mechanism to guarantee a high level of autonomy at the element databases is provided.

To exemplify the thesis, we use a sample application [E92] used to book tours (see Figure 1.2). It includes the booking of accommodation, air tickets, and car rental. The accommodation transaction comprises subtransactions that are used to book a hotel, motel, and/or hostel. The air ticket booking transaction comprises two transactions – one to book a very basic ticket and the other to book a ticket based on the user's carrier preference. The car rental transaction is used to book cars based on the user's preference.

1.3.2 Key Issues

In the past, there have been many contributions and proposals addressing transaction management issues in a MDBS. All these contributions had their own motivation and reasons. However, the key research issues in transaction management in a MDBS addressed by this thesis are:

- Develop an extended transaction model to support transaction management in application-specific MDB environments.
- Identify dependencies between transactions (subtransactions) executing in such environments to study intra-transaction parallelism.
- To guarantee a high degree of autonomy at the underlying databases.
- To provide support for multiple, functionally equivalent, transaction execution alternatives.
- Illustrate how these approaches can be applied to a more general-domain environment.

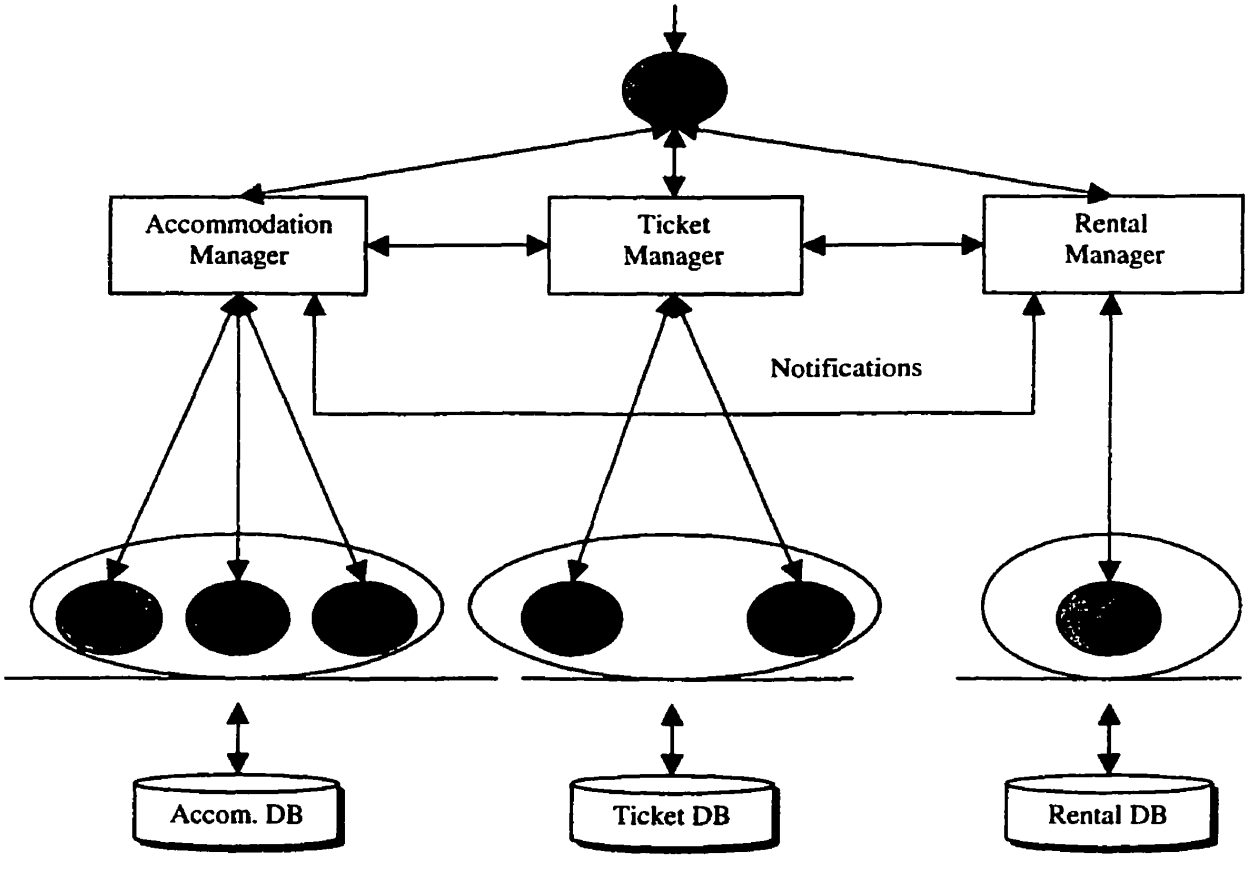

Figure 1.2 Transaction Hierarchies For A Trip Booking Transaction

Extended Transaction Models

Multiple users access a database system concurrently to read and update its data. Such an environment is prone to undesirable situations or inconsistent states if there is no proper

mechanism to interleave the transactions. Concurrency control is the mechanism that aids the proper management of interleavings so there is no interference between transactional operations of different users.

Several approaches have been proposed to enhance the functioning of the concurrency control mechanism in a MDBS. In this thesis, we categorize the past approaches into three generations. Though most of these efforts addressed the transaction management issues in a distributed environment, the same solutions are applicable to the MDB environments.

The first-generation approach was to provide a correctness criterion called *serializability* [BHG87][MRB+92]. This required the scheduling of the transactional operations so that the result of the execution of such a schedule produced the same output and had the same effect on the database as some serial execution of the same operations. However, this approach proved to be too stringent and needed to be relaxed. Further, this was insufficient in the presence of failures. Hadzilacos [H88] proposed the consideration of potential failures that could possibly occur in transaction processing environments. This resulted in a theory of *reliability* as it relates to transaction management. Serializability and reliability together form the first-generation solutions that address the transaction management issues in heterogeneous distributed database environments.

The second-generation approach focused on *relaxing* serializability. Naturally, this approach allowed some inconsistencies in the database resulting in *tolerably* undesirable situations [DE89][B90][BÖ90][PL90][MRK+91][HB96]. Some methods even ignored the integrity constraints of the underlying element databases [DE89]. Despite these efforts, it was realized that the local autonomy in the element databases made it difficult to apply traditional transaction management techniques to MDBSs. Apart from the above-mentioned efforts, Garcia-Molina [G83], Lynch [L83], Farrag and Özsu [FÖ87][FÖ89] address the concurrency control problem using the semantic knowledge of transactions. Alonso *et al.* [AVA+94], Vingralek *et al.* [VHB+98], Schuldt *et al.* [SAS99] attempt to unify the theory of concurrency control and recovery.

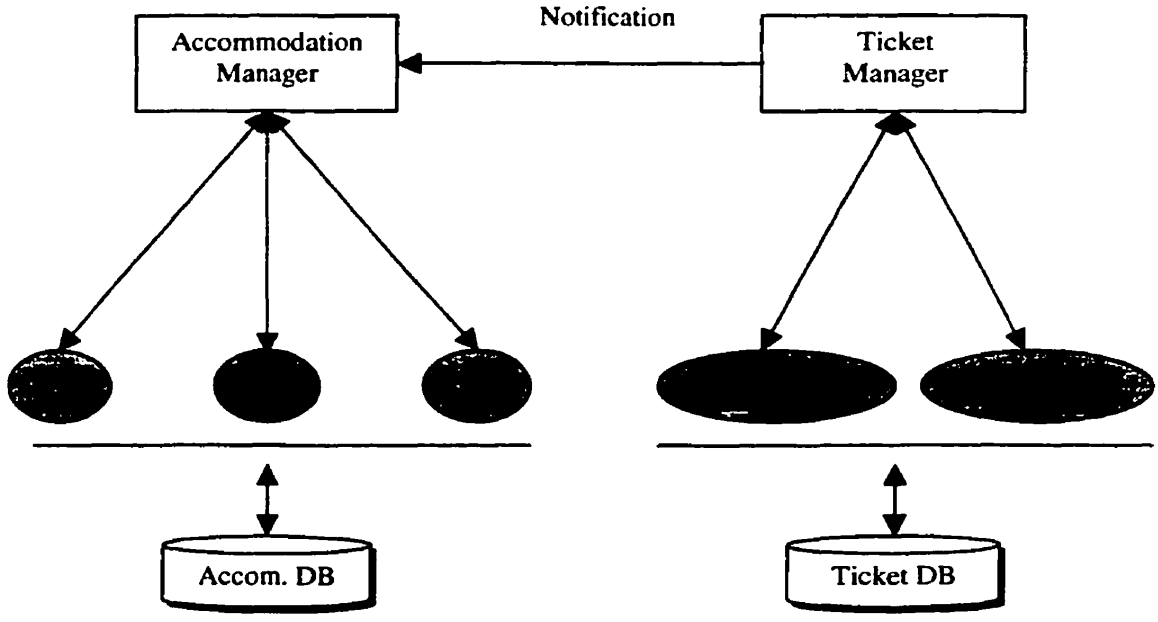

Figure 1.3 Revelation of Partial Results

Several works were proposed to address the inadequacies posed by the traditional transaction processing concepts. Finally, much attention was devoted to the development of *application-specific transaction models*. This resulted in nested transactions [M81][M85], Sagas [GS87], multilevel transactions [W86][W91], *etc.* These models fundamentally disagree with the notion that conflict serializability is the basic correctness criterion. Though the concept of nesting the transactions proved to be an interesting concept, it initially failed to address the autonomy of the element database systems. However, nesting of transactions is useful because it allows controlled concurrency within a transaction and localizes the potential failures. It was based on the nested transaction model that other extended transaction models were proposed. The work presented here uses a novel implementation of the nested transaction in a MDB environment. In this environment, we observe that a transaction is decomposed into a set of subtransactions. Each of these subtransactions is considered to be an operation in the context of other transactions. The subtransactions at the lowest level in the nesting interact with the element databases. The partial results of such interactions are used to determine the execution of other transactions so that they need not wait until the entire

result is available. This referring to partial results is the motivation for identifying the transactional execution dependencies.

One example of a transaction where the partial results can be useful is in a trip booking transaction. In Figure 1.3, we show part of the trip booking transaction shown in Figure 1.2. Consider a trip booking transaction that states that an accommodation may be booked *if* some kind of ticket, an economy ticket or a first-class ticket, is available. The ticket manager spawns the subtransactions for economy and first-class tickets. The outcome of the subtransactions is passed on to the ticket manager. Now, if the economy ticket subtransaction executes and produces a result before the first-class ticket transaction, then the ticket manager immediately lets the accommodation manager know of the result. It does not wait for the outcome (success or failure) of the first-class ticket subtransaction. This lets the accommodation manager spawn the accommodation booking subtransactions without having to wait for any other results from the ticket manager. This type of execution of subtransactions within a transaction enables us to realize the intra-transaction parallelism desired.

Identifying Transactional Dependencies

Another research issue we deal with is the identification of the different kinds of dependencies in an application-specific transaction model. In the case of nested transactions or long-lived multilevel transactions, it can be observed that there is a strong contention for resources among the individual transactions. In such environments, transactions waste a lot of time by waiting for other transactions to finish utilizing the resources they hold. For example, a transaction may have to wait until its sibling's children are in the precommit stage before it can utilize (read/write) their results. Such a nested structure of transactions takes a hierarchical form, where a root transaction has child transactions, which in turn could have children or be a flat transaction.

This thesis argues that a subtransaction can read/write the partial results of another subtransaction that belongs to a totally different parent at any level in the nesting. In such a nested structure, there is less time wasted contending for resources, thereby increasing

the performance. However, the challenge is to identify (the strengths of) the dependencies between the transactions so as to enforce the correct usage of the precommitted results. Hence, based on (the strengths of) the dependencies, the transaction would execute differently. This thesis identifies such dependencies between the subtransactions and categorizes them based on the manner in which they are going to be used. The identification of such dependencies paves the way to study the concurrency control (parallelism) aspects within a transaction. In the environment considered in this thesis, such parallelism is realized at different levels of the transaction model.

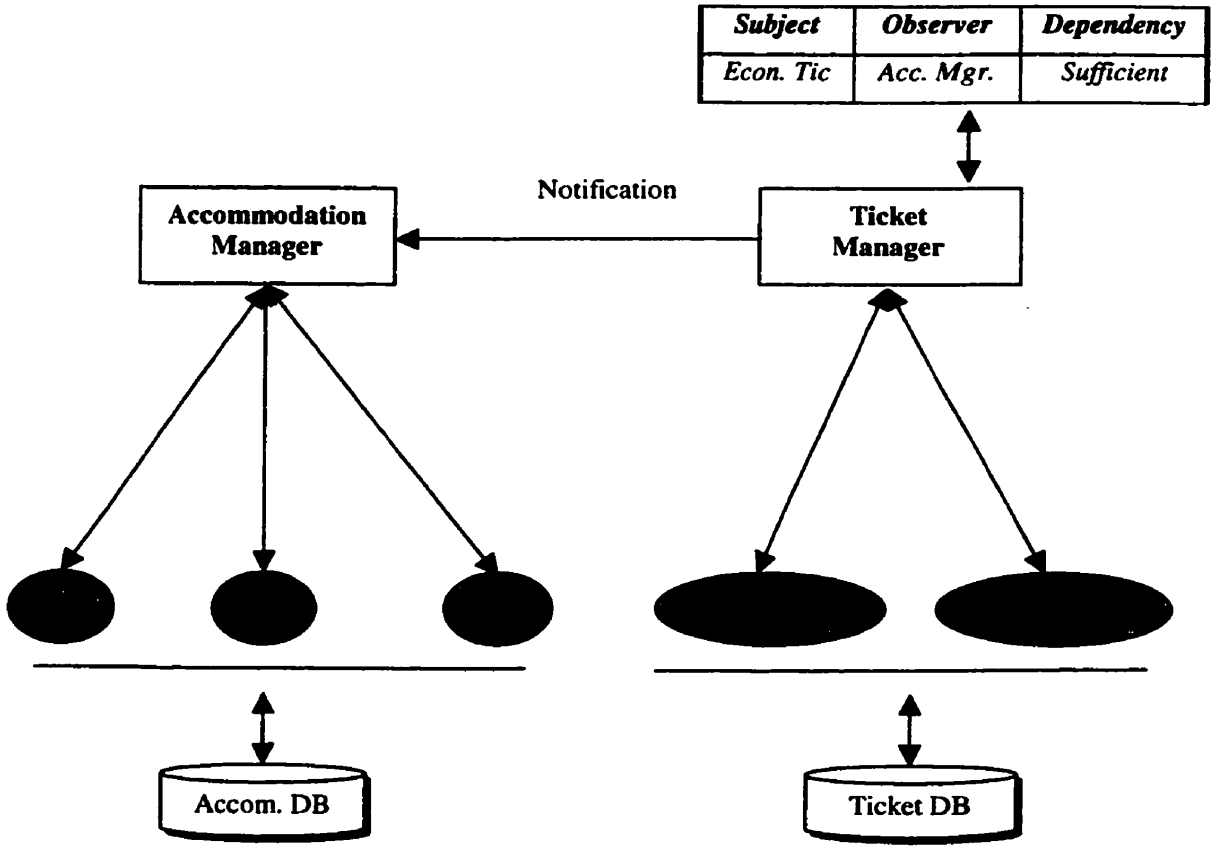

Figure 1.4 Subtransactional Dependencies

Consider a scenario where the dependency relationship at the ticket manager is as shown in the table (execution dependency database) in the top, right corner, of Figure 1.4. The first column in the table indicates the notifying subtransaction (*subject*). That is, the subtransaction that reveals the results. The second column contains the subtransaction that is to be notified (*observer*). That is, the subtransaction that refers to the revealed

result. The third column contains the (strength of) *execution dependency*. In this case, we choose that the availability of an economy ticket is just *sufficient* for the accommodation manager to start any or all of its subtransactions. Hence, upon the positive outcome of the ticket transaction (availability of an economy ticket), the ticket manager checks its dependency database and notifies the accommodation manager about its results. The accommodation manager takes it from there and spawns its subtransactions. The dependencies could broadly be either strong or weak, and based on that appropriate subtransactions are invoked. A complete treatment of such dependencies is the topic of Chapter 3.

1.4 Contributions and Structure of Thesis

In this thesis, we analyze the problems posed by the above issues and implement a solution to address them. This requires the application of a nested transaction model to a typical MDB environment and the capture of dependencies between the subtransactions. Such dependencies are utilized to study the concurrency control aspects within a transaction executing in the framework considered. Due to the nature of the MDB environment and the properties of transactions, there is no single cure-all for the problems mentioned and issues identified above. Specifically, there is no single combination of dependencies that could address the problem. We recognize this and aim at providing a very generic form of transaction management that could result in different processing of the same transaction depending on the combination of dependencies. The thesis is exemplified by the exploitation of the sample application that is used to book online tours. Throughout this thesis we use the same application to explain the different components of this thesis. This application uses the inputs from the user for booking a very basic tour that includes the booking of accommodation, air ticket and car rental. A multidatabase environment is created using a commercial database system. The implementation of the transaction interacts with this environment in the process of booking the tour. The interactions are controlled by the specification of the execution dependencies among several transactions (subtransactions). The results of these transactions vary with the specification of the execution dependencies. The implementation of the subtransaction managers provides an interface mechanism that

characterizes the autonomy level of the underlying system. It must be noted that the implementation of this thesis provides an abstract framework that can be effectively utilized by other advanced database applications, such as, online auctioning, information kiosks, *etc.*

The rest of this dissertation is organized as follows: Chapter 2 provides the necessary background and related work in the area of transaction management. Chapters 3 and 4 form the core of this thesis. Chapter 3 discusses the transactional (subtransactional) dependencies. An application of the new paradigm is presented in Chapter 4. This chapter provides the illustration of an application executing within the abstract transaction framework presented in this dissertation. It also includes a discussion on utilizing the framework to support a more general domain of database applications.

Finally, in Chapter 5, we present a summary of our work and contributions. We present a discussion about the general lessons learned from applying the nested transaction model to a MDBS environment and utilizing the transactional (subtransactional) execution dependencies that exist between them. We conclude Chapter 5, and this dissertation by providing an outline of future research directions.

History is the witness that testifies to the passing of time;
it illumines reality, vitalizes memory, provides guidance
in daily life, and brings us tidings of antiquity.

- Cicero 106-43 BC

Chapter 2
Background and Related Work

This chapter introduces the reader to the necessary background and related research work in transaction management in multidatabase systems. Section 2.1 provides an introduction to multidatabase architecture. In Section 2.2, we discuss transaction management in multidatabase systems. Sections 2.3 and 2.4 present the three generations of research in this area. The leading open research questions are the subject of discussion in Section 2.5. Section 2.6 concludes this chapter with a summary.

2.1 Multidatabase Architecture

Businesses around the world rely on a very wide range of information sources to conduct their everyday chores. These sources of information are usually databases whose size and structure depends on the size and type of the business. These databases grow with the businesses and hence the information is accessed from several nodes. Each node uses a copy of the whole database. The existing data is distributed geographically based on specific needs. Such a distribution of data results in a situation where the same data is stored in dissimilar platforms and dissimilar languages access them. Though these

dissimilarities pose problems, they enable the sharing of information among geographically distributed databases and users. A user of these databases believes only a single centralized database is accessed. Such databases are called distributed database systems.

A *distributed database system (DDBS)* is an information system composed of a networked collection of multiple databases that are logically interrelated (see Figure 2.1). A *distributed database management system (DDBMS)* is a software facility that permits the management of the DDBS and makes the distribution transparent to the users [ÖV99].

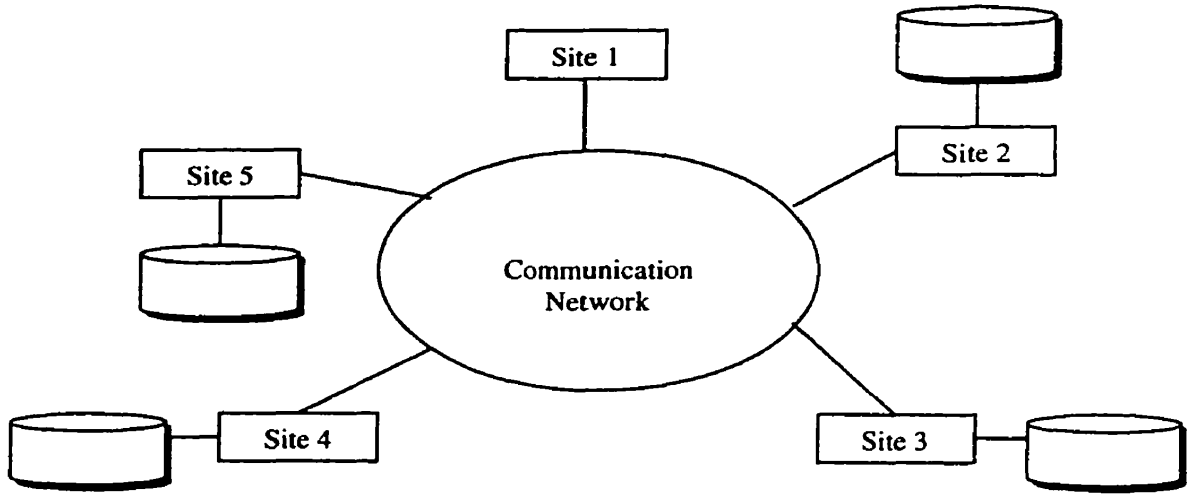

Figure 2.1 DDBS Environment (Courtesy: [ÖV99])

A *multidatabase system (MDBS)* is a special case of the distributed database system. It is an interconnection of multiple databases that are characterized by autonomy and heterogeneity. A *multidatabase management system (MDBMS)* is a software facility that coordinates access to the underlying databases. Figure 2.2 depicts a high-level architecture of a multidatabase system.

The MDBMS is the core component of multidatabase architecture. This is responsible for the correct execution of the transactions submitted to it. Each participating database has its own database management system called the *local database management system*

(LDBMS). This is responsible for the correct execution of the local transactions submitted to it. The MDBMS has no control over the execution of transactions at the element databases. Hence the element databases are characterized by some degree of autonomy. Further, each of these databases may have their own data model and access languages. Hence these databases may also be characterized by heterogeneity.

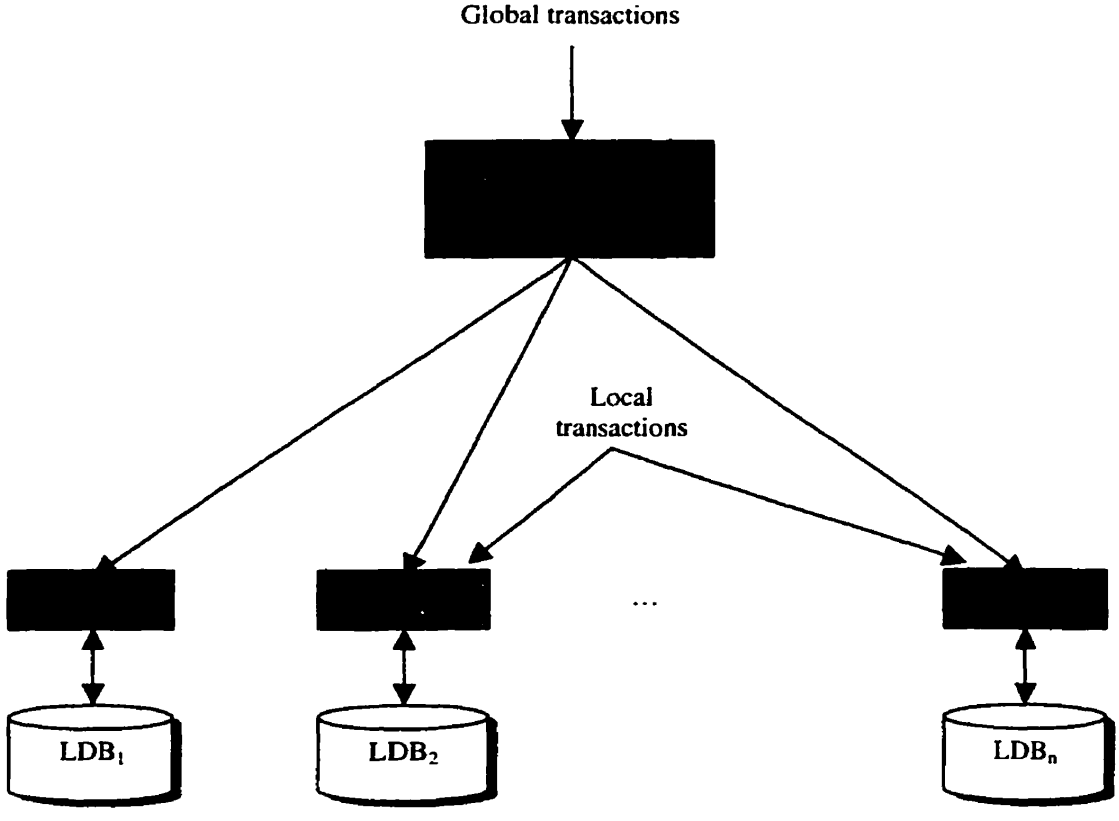

Figure 2.2 A Multidatabase Architecture

2.1.1 Definitions: [BE99]

This section provides a brief description of the key definitions used throughout the balance of this thesis.

MDBS: An MDBS is composed of a set of local databases *(LDB = {LDB_1, LDB_2, ... LDB_n})*, with each managed by its own corresponding local database management system *(LDBMS = {$LDBMS_1$, $LDBMS_2$, ... $LDBMS_n$})*. ■

Local Transactions: A set of transactions submitted to $LDBMS_i$ ($LT_i = \{LT^1_i, LT^2_i, \ldots LT^n_i\}$) is called local transactions. $LDBMS_i$ is responsible for all the local transactions submitted to it. ■

Global Transactions: A set of transactions submitted to the MDBMS ($GT = \{GT_1, GT_2, \ldots GT_n\}$) is called global transactions. ■

Each of these global transactions (GT_j) is decomposed into a set of global subtransactions.

Global subtransactions: A set of transactions obtained from the decomposition of a global transaction. $GT_j = \{GST^{i1}_{j1}, GST^{i2}_{j2} \ldots GST^{im}_{jp}\}$, where each GST is submitted to its corresponding LDBMS. The superscripts in the above set identify the LDBMSs. ■

2.2 Transaction Management

Guaranteeing correct execution of transactions over the MDBS has been the focus of research over the past 15 years. Initial research in this area yielded multidatabase management systems, while lately the focus has been on addressing transaction management issues in specific application domains. Nevertheless, the questions framed in the initial stages of transaction management research have not yet been answered completely.

Figure 2.2 presented an architectural model of multidatabase systems with the depiction of local and global transactions. Transactions submitted to the MDBMS are called *global transactions*. These are decomposed into *global subtransactions* that are then directed to specific databases where the corresponding data is located. Transactions posed directly against the local databases are *local transactions*. These transactions execute under the control of the local DBMS. Hence the MDBMS has no control over them. This lack of control poses a key challenge to transaction management in a MDBS. Traditionally transactions are characterized by a need to support the ACID properties.

MDBS transaction management is particularly challenging due to the autonomous nature of the components in the environment. There are different types of autonomies identified by various researchers [SL90][B94]. Further, transaction management is complicated by the heterogeneous nature of the element databases. Several solutions have been proposed to address the problem with these issues. Earlier research focused on the heterogeneous nature of the environment while later the focus shifted towards the autonomous nature of the underlying system [BE99]. However, research in these two areas led to several solutions that address the transaction management issues in MDBSs. Lately, the focus is on providing transaction models based on application domains. This led to the development of several transaction models, a discussion of which is available in Elmagarmid's work [E92].

2.3 Concurrency Control and Reliability

Past research considered the transaction management problem in two orthogonal dimensions – *serializability* [BHG87][MRB+92] and *reliability* [H88]. Serializability serves as a correctness criterion to the concurrency control algorithms executing in fault-free environments. It is supported by the atomicity and isolation properties of transactions. Reliability, in addition to serializability, is a correctness criterion for concurrency control algorithms executing in fault-prone environments. It guarantees the execution of transactions and persistence of their results. Reliability is guaranteed by the consistency and durability properties of transactions. The concurrency control algorithms are implemented using *locking* or *timestamp ordering* concepts [BHG87]. This section presents the various correctness criteria proposed in the past. It is divided into two sections that deal with the first-generation and second-generation solutions, respectively. The results discussed here have their own versions of concurrency control algorithms broadly based on either locking concepts or timestamp ordering concepts.

2.3.1 First-generation Solutions

The notion and development of transaction models supporting serializability form the first generation of research in this area [BHG87]. The primary objective is to achieve global serializability. When two or more transactions execute concurrently, operations of

one transaction may execute between the operations of another transaction. This execution, known as interleaved execution, may lead to incorrect behavior in the transactions. This eventually leads to undesirable outcomes or inconsistent database states. This is called the *interference problem* [BHG87]. It can be avoided by not allowing the transactions to interleave at all. The resulting execution where no two transactions interleave with each other is called a *serial* execution. The transaction system must ensure the execution occurs so that all the operations of one transaction precede all the operations of another transaction. However, from the user's perspective both transactions execute atomically. For example, consider a banking transaction involving the transfer of money from one account to another, as shown in Figure 2.3(a).

```
Begin Transaction Transfer
        Start;
        Withdraw (1234, $100);
        Deposit (4321, $100);
        Commit;
End Transaction
```

Figure 2.3(a)

```
Begin Transaction Transfer
        Start;
        Withdraw (1234, $100);
        Withdraw (1234, $200);
        Deposit (4321, $100);
        Commit;
End Transaction
```

Figure 2.3(b)

Figure 2.3 A Transfer Transaction

The transfer transaction involves two operations – *withdraw()* and *deposit()*, each of which receive the account number and amount as parameters. Let the account number from which the money is transferred be 1234 and the account to which it is transferred be 4321. Let the initial amount in account 1234 be $200. The result of the execution of the transaction shown in Figure 2.3(a) would be a debit of $100 from account 1234. Hence the balance in it becomes $100. Suppose there is a withdraw operation (of another transaction) occurring on the same account number for $200. The transaction appears as shown in Figure 2.3(b). Its execution results in debiting an account that is already in an undesirable state (negative money!). A similar situation occurs to the credited account if, for example, a withdraw operation occurs on it before the transfer transaction commits.

Serializability

Bernstein *et al.* [BHG87] describes three forms of serializability namely, *Conflict Serializability, View Serializability and Final-state Serializability.* Serializability theory presents the concurrent execution of a set of transactions using the concept of *histories.* A history is comprised of a set of read and/or write operations. These operations may or may not be conflicting. Two operations are *conflicting* if they operate on the same data item and at least one of them is a write operation. There are two types of histories in a multidatabase environment– *local* and *global* – representing the execution at the local and global levels of the MDBS architecture respectively (see Figure 2.2 on Page 17).

According to conflict serializability theory, an execution is serializable if it is conflict equivalent to a serial execution of the same transaction. A history comprising such an execution is called a *conflict serializable history.* The histories are analyzed by representing their executions in the form of a directed graph called a *serialization graph.* The nodes of a serialization graph are the transactions and its edges define the ordering of the operations of the transactions. The *serializability theorem* states that a history is serializable *iff* its serialization graph is *acyclic.* Two histories are said to be *equivalent* if both are defined over the same set of transactions, both have the same operations, and they order the conflicting operations of non-aborted transactions in the same way.

View serializability is defined in the same terms as conflict serializability. Hence, a history is said to be view serializable, if it is view equivalent to some serial history. However, these two serializabilities are totally different. Bernstein *et al.* [BHG87] shows that a conflict serializable history is contained in view serializable. For all practical purposes earlier research used conflict serializability instead of view serializability as the concurrency control correctness criterion because of the requirement to maintain the ACIDity of concurrent transactions.

The concurrency control methods are generally classified into two types – *optimistic* and *pessimistic.* Optimistic methods assume that, not many transactions conflict with each other whereas pessimistic methods believe that, most of the transactions conflict with

each other [ÖV99]. The functional differences between these methods lies in when they synchronize the execution of concurrent transactions.

Reliability

Research on serializability as the correctness criterion typically assumes that the transaction processing occurs in a fault-free or fault-tolerant environment [H88]. Practically this is not the case because there are numerous sources of failures. Examples include transaction, system, or media failures as reported by Gray [G81]. Hence the assumption that all the transactions would complete correctly and produce a consistent state cannot be substantiated.

A *transaction failure* occurs if the transaction is interrupted before all its operations are processed. Consider the transfer transaction example shown in Figure 2.3(a) on Page 20. Suppose the transaction fails after the withdraw operation but before the deposit operation. This will result in an inconsistent database state because the money withdrawn is recorded while the deposit is not recorded (lost money!). Such transactions must be aborted thereby undoing all the operations. This is a major focus of the reliability aspect of transaction management.

A *system failure* occurs due to system crashes or loss of information from the volatile storage media. In such unforeseen circumstances, the transaction execution must be aborted and the effects of the committed transactions must be undone. In some cases, the states of the transactions are saved so that, on recovery, the transactions could be rolled back to a previous correct state. When a portion of the stable storage media is lost, a *media failure* occurs. These are the other areas that are addressed by the reliability mechanisms.

Based on the notion that serializability is an insufficient correctness criterion in a fault-prone environment, Hadzilacos [H88] proposed a correctness criterion that has three dimensions to it. An execution is correct if, at any time,

1) the committed transactions have been processed in a serializable fashion (*Commit serializable*[*]),
2) any uncommitted transaction can be aborted without invalidating the semantics of committed ones (*Recoverable*), and
3) the "correct" database state can always be reconstructed from information stored in stable storage (*Resilient*)

Commit serializability is a modification of traditional serializability applied only to the committed transactions in an execution. This requires the committed transactions in an execution be serializable. It ignores the transactions that are obliterated and hence applies serializability only to those that committed or run to completion. This notion can be applied to conflict and view serializability.

The *Recoverability* notion states that the abortion of the uncommitted transactions does not affect the semantics of the committed transactions. This notion is closely associated with the durability property of transactions. The results of all committed transactions must be made permanent while those of the uncommitted (aborted) transactions must be obliterated. Recoverability has a direct application when cascading aborts need to be addressed. Cascading aborts occur when a transaction refers to the results of a transaction that has aborted. Though the recoverability notion is evidently powerful it can be observed that it is stringent too.

Resiliency refers to the ability of the system to reconstruct the database to a correct state by using the information stored in the stable storage in case of a system failure. This works when a system failure occurs but not for a media failure. However resiliency depends on the choice of the storage media. For instance, if the environment depends on the information in the volatile storage for its reliability (correctness), resiliency algorithms utilize the volatile storage to restore the database to a consistent state. But in

[*] *'Commit Serializable', 'Recoverable' and 'Resilient' are the terms used in [H88]*

most cases, reliability information is stored in the stable storage since it survives system failures.

Bernstein *et al.* [BHG87] describes reliability using histories. They formulate three types of histories – *recoverable (RC), avoids cascade aborts (ACA) and strict (ST).*

A history is *recoverable* if each transaction commits after the commitment of all transactions from which it read.

A history *avoids cascade aborts* if a transaction reads only those values that are written by any committed transaction or by itself.

A history is *strict* if a data item can be written only after the transaction that previously wrote into it terminates (either commits or aborts).

The concept of *recoverability* by Hadzilacos [H88] is related to the concepts of *RC* and *ACA* presented by Bernstein *et al.* [BHG87] Further, *RC*, *ACA*, and *ST* are similar to *commit-serializability* discussed by Hadzilacos. Bernstein *et al.* describes the above in terms of *prefix commit-closed* property.

It is evident from Bernstein *et al.* [BHG87] and Hadzilacos [H88] that the correctness criterion for the concurrency control algorithms must consider not only serializability, but also recoverability. Most research that followed either ignored the reliability aspects or attempted to unify the theory of concurrency control and recovery. This thesis assumes the environment to be fault-free and hence we too are not concerned about the reliability aspects of transaction management.

2.3.2 Second-generation Solutions

It was found that many computer applications required a less stringent form of concurrency control mechanism than the one supported by conflict serializability. This resulted in several works addressing the concurrency control issue by relaxing the notion

of conflict serializability through the exploitation of serializability at the global and local levels [DE89][B90][BÖ90][PL90][MRK+91][HB96]. Garcia-Molina [G83], Lynch [L83], and Farrag and Özsu [FÖ89] address the concurrency control problem using the semantic knowledge of transactions. Alonso *et al.* [AVA+94], Vingralek *et al.* [VHB+98], and Schuldt *et al.* [SAS99] attempt to unify the theory of concurrency control and recovery.

Quasi Serializability

Du and Elmagarmid [DE89] introduced Quasi Serializability (QSR) as a correctness criterion for concurrency control in heterogeneous database systems. This was primarily based on the notion that a heterogeneous database system is hierarchical in nature due to the autonomy of element databases and thus maintaining global serializability is very difficult. The objective of the effort was to provide a correctness criterion for global concurrency control. Further, it realized that two global transactions that do not reference common data items could also conflict. These conflicts are called *indirect conflicts*. QSR assumes at most one subtransaction executes at each local site.

The correctness of an execution in QSR is based on the notion of a *quasi-serial history*. A quasi-serial history indicates that only the global transactions are executed in a serial fashion. A history is quasi-serial if:

1) all the local histories are (conflict) serializable, and
2) there exists a total order of all global transactions so that for every two global transactions, G_i and G_j, G_i precedes G_j in the order and all G_i's operations precede G_j's operations in all the local histories in which they appear.

A history is *quasi-serializable* if it is (conflict) equivalent to a quasi-serial history. All the local histories in a quasi-serializable history are serializable. Additionally, global transactions are executed in a serializable fashion.

The QSR histories are characterized using *Quasi Serialization Graphs (QSG)*. The *Quasi Serializability theorem* states that a global history is quasi-serializable *iff* all the local histories are (conflict) serializable and the *QSG* for that global history is *acyclic*.

The environment model considered by Du and Elmagarmid guaranteeing QSR is restricted as follows:

1) It must not have any intersite integrity constraints.
2) A global transaction executing in a site is independent of its execution at other sites.

Multidatabase Serializability

Barker and Özsu [BÖ90] introduced Multidatabase Serializability (MDBSR). This work is similar to Du and Elmagarmid's [DE89] QSR because both these approaches consider serializability at local and global levels. MDBSR captures serializations at the local histories and the history of transactions that are not completely contained at a single DBMS. This work is different from QSR because it considers the importance of the reliability aspects of transaction management and hence considers transaction management more completely than QSR. The central aspect of this study is that heterogeneity is orthogonal to autonomy at the element database level. MDBSR considers complete local autonomy. As a direct consequence of the assumption about local autonomy, there are no value dependencies between data stored in different databases. Barker and Özsu do not consider any replication of data whatsoever.

The correctness of the execution of transactions is based on the notion of a *MDBSR history*. A history is *MDB-serial* if (similar to QSR-serial):

1) all the local histories are (conflict) serializable, and
2) there exists a total order of all global transactions so that for every two global transactions, G_i and G_j, G_i precedes G_j in the order and all G_i's operations precede G_j's operations in all the local histories in which they appear.

A MDBSR history is considered to be *MDB-Serializable iff* they are defined over the same set of transactions and they order conflicting non-aborted operations the same way [BÖ90].

The MDB-serial histories are analyzed using a variant of the serialization graph by Bernstein *et al.* [BHG87]. It is called the *Multidatabase Serializability Graph (MSG).* The *Multidatabase Serializability theorem* states that a history is MDB-Serializable *iff* it's *MSG* is *acyclic* [BÖ90].

Epsilon Serializability

Pu and Leff [PL90][PL91][PL92] proposed Epsilon Serializability (ESR) as a correctness criterion for concurrency control. This is a generalization of traditional serializability for specific application domains. The purpose of this correctness criterion is to explicitly allow a limited amount of inconsistency in transaction processing. The algorithms guaranteeing ESR are called the *Divergence Control (DC) methods* [WYP97]. These are the equivalents of concurrency control methods ensuring traditional serializability.

The *Epsilon Transactions (ET)* [P91] are classified into queries (Q^{ET}), updates (U^{ET}) and regular transactions. The Q^{ET}s have read operations while any *ET* with at least one write operation is a U^{ET}. The transaction processing system identifies the difference between an initial database state (u) and a final database state (w) after the execution of the *ETs*. This difference is denoted by ε, which is the amount of inconsistency. If the value is greater than '*0*' or equal to an arbitrary value '*e*' (this is the limit), then an *ESR log* (history) is created. This is equivalent to the *SR log* (history) of traditional serializability.

A history in this framework is called an *ET-wise ESR log*. The algorithms take two units called import and export units as their inputs. A Q^{ET} imports some inconsistency, while a U^{ET} exports some inconsistency. If the limits of the inconsistency imported (*ImpLimit*) and exported (*ExpLimit*) are greater than zero, the database may degenerate and become inconsistent with no bounds. The following table classifies the *ETs*:

	ImpLimit = 0	ImpLimit > 0
ExpLimit = 0	Transaction	Q^{ET}
ExpLimit > 0	U^{ET}	Unbounded Inconsistency

Table 2.1 Epsilon Transaction Classifications

The DC algorithms employ inconsistency counters to detect the inconsistencies. An extension of the two-phase commit algorithm, the DC algorithm either allows or disallows the *ETs* to proceed depending on the import/export inconsistency counters.

The advantages of ESR are [P91]:

1) It is a general framework, applicable to a wide range of application semantics,
2) It is upward compatible, since it reduces to conflict serializability when $e \rightarrow 0$, and
3) It has a large number of efficient supporting algorithms.

ESR, as mentioned earlier, is only suitable for environments that tolerate a limited amount of data inconsistency. Further, the database state space must have all the properties of a metric space. Hence this is suitable only for numerical data items, and not string data items. Though the authors claim that ESR can be extended to support string data items the literature does not cite any such example. Since ESR finds application in environments where a certain amount of inconsistency is tolerated, the amount of inconsistency must be known in advance.

Ramamritham and Pu [RP95] formally characterize ESR. A quantification of the inconsistency bounds imported by the *ET* is presented. They also examine how to ensure

that only epsilon serializable histories are produced. Finally, they examine how the inconsistency read by an *ET* percolates to the results of the query.

Two-level Serializability

Mehrotra *et al.* [MRK+91] introduce two-level serializability (2LSR) as a correctness criterion for concurrency control in heterogeneous distributed database environments. It attempts to relax global serializability.

2LSR requires the projection of the global schedule on the set of global transactions to be serializable and each of the local schedules to be serializable as well. The environment model considered here is the result of the integration of various preexisting databases. Though these databases do not have any integrity constraints when considered individually, their integration introduces intersite integrity constraints. This makes it different from the environment model considered by Du and Elmagarmid [DE89]. Since 2LSR requires the projection of a global schedule on a set of global transactions to be serializable and each of the local schedules to be serializable, it can be shown that 2LSR schedules are not always serializable [CR99]. 2LSR schedules preserve database consistency by exploiting the knowledge about the nature of the intersite integrity constraints. Partitioning the data items into two disjoint sets namely, global and local data items, aids in exploiting this knowledge. Hence, 2LSR schedules preserve database consistency only in certain environment models where the intersite integrity constraints are known.

Transaction Processing Using Semantics

While serializability is the correctness criterion guaranteeing database consistency in the presence of syntactic information, it can be weakened to enhance the level of concurrency when semantic information is available. This is the motivation for research proposed by Garcia-Molina [G83], Lynch [L83], and Farrag and Özsu [FÖ89]. However, not all applications have the semantic information of transactions. Hence, transaction processing using semantics is not possible in all applications.

Garcia-Molina [G83] proposed the notion of *semantically consistent schedules* and *sensitive transactions* to address the problem of transaction processing in distributed databases. A sensitive transaction outputs the data that are seen by the users and those data must be based on a consistent database state. A schedule is classified as semantically consistent schedule if its execution transforms the database state to a consistent state and all the sensitive transactions obtain a consistent view of the database. The transactions are classified into a collection of disjoint classes (*compatibility sets*). All the transactions that belong to the same class are categorized, as *compatible* transactions while the rest are *incompatible* transactions. The compatible transactions can interleave arbitrarily while the incompatible transactions cannot. This allows two extreme levels of interleaving among the transactions. The users specify the semantics by designing their own transaction processing mechanisms wherein they incorporate the necessary knowledge for interleaving the actions of the transactions without violating consistency. Since this is a cumbersome process for the user, Garcia-Molina suggests the use of a transaction processing system that accepts the "rules" of the most common semantically consistent schedules.

Lynch [L83] weakens the notion of serializability by permitting controlled interleaving among transactions. This weaker notion of correctness criterion is referred to as *multilevel atomicity.* Multilevel atomicity supports different views of atomicity for the same transaction when viewed by different transactions. This finds use in environments where the transaction processing is inherently hierarchical, possibly due to the hierarchical nature of the organization.

A set of operations is grouped together to form a transaction unit. This grouping is done for at least three different purposes: (1) to make the operations of a transaction (a logical unit) persistent, (2) define atomicity and thereby serializability, and (3) use the grouping as a unit of recovery. Lynch argues the use of different units for each purpose mentioned above. First, the logical unit must be as large as possible. Since this poses a strong serializability requirement, another mechanism is superimposed on the transaction mechanism to define atomicity. Hence, the second argument is: the unit of atomicity must

be as small as possible for maximum concurrency. Third, the unit of recovery can be anywhere in between. Lynch uses the concept of *breakpoints* between such long transactions as the point where other transactions interleave.

For example, consider a money transfer transaction in a banking application [L83]. Transfer transactions might be allowed to interleave arbitrarily with other transfer transactions. However, a different type of transaction, for instance, an audit transaction that returns the total amount in an account, cannot interleave between the transfer transactions. That is, the entire audit transaction gets an atomic view of the entire transfer transaction and vice versa. Hence, a transfer transaction will have a set of breakpoints for other transfer transactions and another set of breakpoints for audit transactions.

Lynch allows many possible interleavings between the range specified by Garcia-Molina. That is, between one extreme where it allows only serializable interleavings and the other extreme where the interleavings are unconstrained. The steps of a transaction occurring between two breakpoints always occur atomically at least from the user's perspective. However, if there are breakpoints only in the beginning and end of the transaction, then this reduces to the requirement of traditional serializability. Many other cases are also possible depending on where the breakpoints occur. By using breakpoints instead of compatibility sets, several levels of compatibilities among transactions are defined. This structures the levels of compatibilities in a hierarchical manner where the interleavings at a higher level encompasses those at the lower levels.

Farrag and Özsu [FÖ89] use the concept of breakpoints and exploit the use of semantics for transaction processing. This work differs from Garcia-Molina's in that, it does not use the compatibility sets. It differs from Lynch's because it does not require the interleavings to be hierarchical. They specify the notion of consistency by describing the allowable interleavings among the transactions that are safe to execute and then ensuring that each schedule thus produced is equivalent to a correct schedule. The allowable interleavings are specified at each breakpoint depending on the application needs. This supports the concept of multilevel atomicity [L83]. However, it is different because the

interleavings specified at one level does not include the interleavings specified at the lower levels. Hence, it does not require the interleavings to be hierarchical.

Farrag and Özsu introduce a new class of schedules called *relatively consistent (RC) schedules*. An *RC schedule* has an acyclic precedence graph (serialization graph). A topological sort of that graph yields a correct schedule. This class of schedules contains both serializable and nonserializable schedules. A lock-based concurrency control mechanism is presented that produces only *RC schedules*.

Though considering the semantics of transactions for transaction processing is an interesting concept, there are problems associated with it [G83]. First, it is difficult for the transaction processing mechanism to identify the schedules that are semantically consistent. Even if the transaction processing mechanism is provided with the information about the consistency constraints, there is no way for it to know the semantic consistency of the schedules before running them on the database. Second, it may be impossible to obtain the results of a semantically consistent schedule with any schedule that is serializable. Further, this may be undesirable to some users. Third, the user has to specify the consistency constraints to the transaction processing system.

Unifying Models

While there are research efforts in the area of providing concurrency control correctness criterion, a different school of thought is attempting to unify the theories of correctness criteria and reliability [SWY93][AVA+94][LHL97][SAS99]. This theory unifies atomicity and isolation into a common framework to avoid the shortcomings when considering them as orthogonal problems. Schek *et al.* [SWY93] and Alonso *et al.* [AVA+94] introduce the notion of (prefix-) expanded serializability and (prefix-) reducibility for their unified model of correctness criterion. Lee *et al.* [LHL97] introduce a unified approach to global concurrency control and recovery in the MDBS environment. Similar to other works in this area, Lee *et al.* [LHL97] do not consider the problems of serializability and reliability as two orthogonal concepts. They propose the notion of *rigid conflict serializability (R-SR)* that ensures serializability in a distributed,

fault-free system. They address the recovery aspect of correctness criteria in a fault-prone system by developing a *context-sensitive and late redo recovery scheme*.

One of the more recent works in this area is by Schuldt *et al* [SAS99]. This work applies the unified theory to address the transaction process management problem where not all the activities are compensatable and where more generalized transaction properties are applicable. An example of such a transaction processing system is Computer Integrated Manufacturing (CIM) [SAS99]. The unified theory of concurrency control and recovery finds its application in the areas of electronic commerce, workflow systems, and other systems that involve many subsystem-level processes that are also transactional. In this thesis, concurrency control and recovery are considered as orthogonal problems and we focus on the concurrency control aspects. Hence, we do not delve further into the unified theory of concurrency control and recovery.

2.4 Advanced Transaction Models / Formalism

The first- and second-generation approaches mainly aimed at providing correctness criterion for a generalized transaction management system. These approaches gave way to numerous results that are still being used in many transactional environments. For example, conflict serializability is still the most used correctness criterion in all commercial transaction managers. Due to the various environments encountered, there exists a need for advanced transaction models, which will increase as e-business grows. These environments have motivated the development of extended or advanced transaction models. We refer to this phase of research as the third-generation approach. The following are some reasons why advanced transaction models and new correctness criteria have been proposed [WS92]:

1) To provide better support for long-lived activities in advanced database applications. For example, the daily batch update transactions in a banking application or an insurance claims transaction.
2) To relax the classical ACID paradigm thereby providing more flexibility as to when updates are made visible to concurrent transactions. Most advanced

applications require a less restrictive mechanism of transaction management. For instance, in a trip-booking application, strict ACID properties could have adverse effects on the performance of the application.

3) To capture more semantics of transactional operations in advanced applications. Capturing semantics of the application aids in better management of the transactions. For example, in long-lived transactions, capturing the semantics of the application enables the transactions to access the needed resources with less contention.
4) To enhance (inter-/intra-) transaction parallelism. Extending the concepts of transactions based on the application requirement aids in enhancing the concurrency aspects of the transactions.
5) To deal with multiple autonomous subsystems in a federated environment. For example, a nested transaction model can be easily mapped onto a MDB environment characterized by its autonomy.

Most MDBSs use the concurrency control and recovery algorithms mentioned in Section 2.3 (specifically, the first- and second-generation results). However, with the advent of advanced transaction models, an imperative need to apply these models to the MDBS was realized. Many such transaction models have been applied to MDB environments with varying degrees of success [BE99]. This section presents an overview of past research in the area of advanced transaction models.

2.4.1 Nested Transaction Model

This was the first advanced transaction model proposed [R78][M81][M85]. A nested transaction is one that is divided into subtransactions each of which are either divided further or is composed of only atomic operations (see Figure 2.4). This modeling of transactions gives rise to a hierarchy of transactions comprising a top-level transaction, subtransactions and leaf level transactions. The transactions at the leaf level are flat transactions and they are the only ones that interact with the data sources [M81][M85]. The higher-level transactions organize the transaction execution flow to invoke the subtransactions. The subtransactions' execution is made visible only to the parent after

the subtransactions reach their *precommit* stage. The siblings can never access the changes made by other subtransactions. The subtransactions do not necessarily adhere to the ACID paradigm unlike the top-level transaction. Hence, the subtransactions may be atomic, isolated and consistent but not durable until the top-level transaction commits. Further, the subtransactions commit only after the top-level transaction commits. Until then, they remain in the precommit stage waiting for the top-level transaction to commit. If the top-level transaction aborts for some reason, all the work done by the subtransactions are aborted as well. Hence the durability of the subtransaction is observed only upon the commitment of the top-level transaction. Such a nested model is called a *closed nested transaction model.*

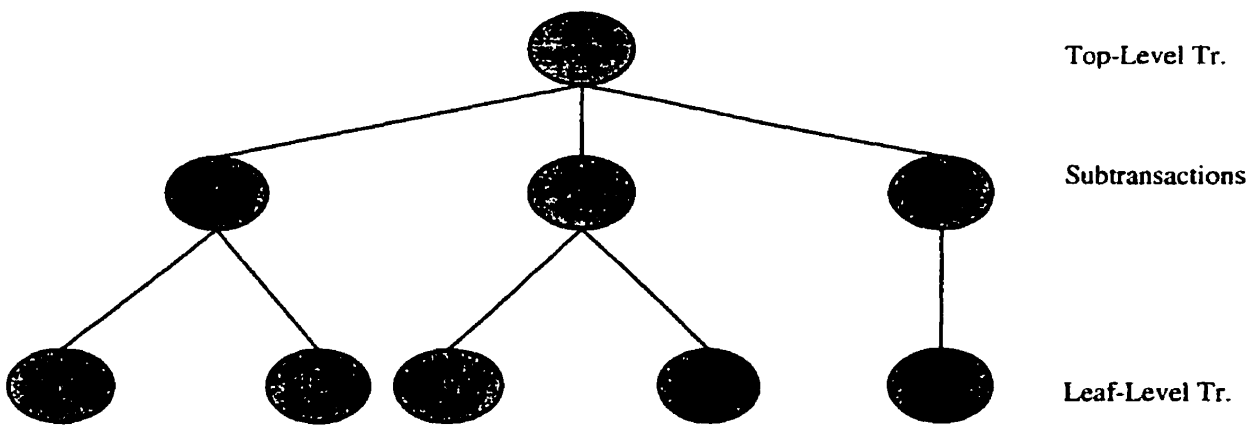

Figure 2.4 Nested Transaction Model

Most advanced models are based on the principles of the nested transaction model introduced by Reed [R78]. The nested transaction model introduced by Moss [M81] [M85] addresses its application in distributed computing and hence is of great importance to research in the area of transaction management. When this model is applied to a MDB environment it is observed that the global transaction and the global subtransactions of an MDBS superimpose the top-level transaction and subtransactions of a nested transaction respectively. If it is a failure-prone MDB environment, then the nested transactions must provide recovery mechanisms through, for instance, *compensating transactions.*

The nested transaction model is appealing in many ways. The division of a transaction into subtransactions provides the following advantages:

1) Enhances modularity of the transaction,
2) Enhances intra-transaction parallelism, and
3) Localizes potential failures.

The application of this model to an MDBS provides interesting insights and is an area under constant research.

2.4.2 Multilevel Transaction Model

Multilevel transactions are special cases of nested transactions in which operations at a particular level are implemented by operations of some lower level of abstraction [W86][W91]. A multilevel transaction in a system with n levels $L_0, L_1, \ldots L_{(n-1)}$, is defined as a tree of height $n+1$ such that all leaf nodes are at the same level, L_0. The nodes of the tree are called actions that represent executions of level-specific operations [WS92]. Whenever two transactions commute, their execution sequence does not matter. Such situations that occur among the subtransactions are captured by multilevel transactions using a layered hierarchy. That is, multilevel transactions are nested transactions in a layered database environment.

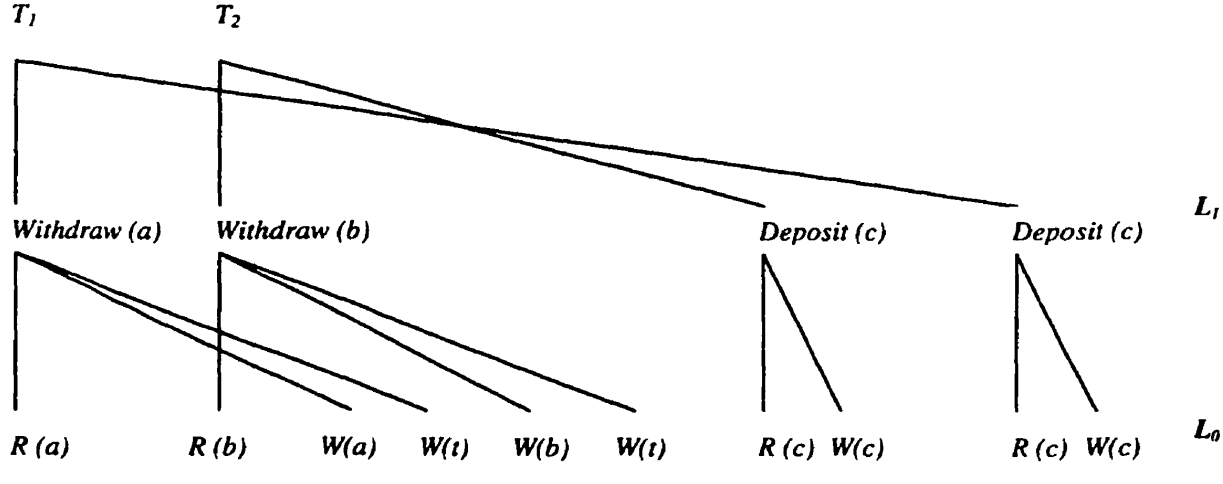

Figure 2.5 Concurrent Executions of Multilevel Transactions (Courtesy: [WS92])

In a multilevel transaction as shown in Figure 2.5, a concurrency control mechanism is needed at all levels so that each higher level (non-conflicting) operation is executed in an indivisible manner. However, the conflicts among operations at the lower levels have to be addressed. The goal of the L_i-level concurrency control is to isolate the $L_{(i+1)}$ subtransactions from each other. Hence, the high-level operations are executed as subtransactions that usually follow a general concurrency control strategy. Recovery is made possible using compensating transactions. For example, in Figure 2.5, aborting T_2 after T_1 has committed requires two compensating transactions – *withdraw (c)* and *Deposit (b)*. A method of multilevel recovery requires that the transactions are atomic and persistent and the subtransactions are atomic as well [WS92]. Further, during a restart, a redo must be performed at the bottom level L_0. The compensating transaction can be executed in the same framework as concurrency control by treating them as additional regular expressions.

This model finds suitable application in an MDBS because it offers a high degree of autonomy to the element databases and provides global consistency [BE99]. Though the application of this model to MDBS is interesting, defining commutativity of the subtransactions in such an environment is difficult.

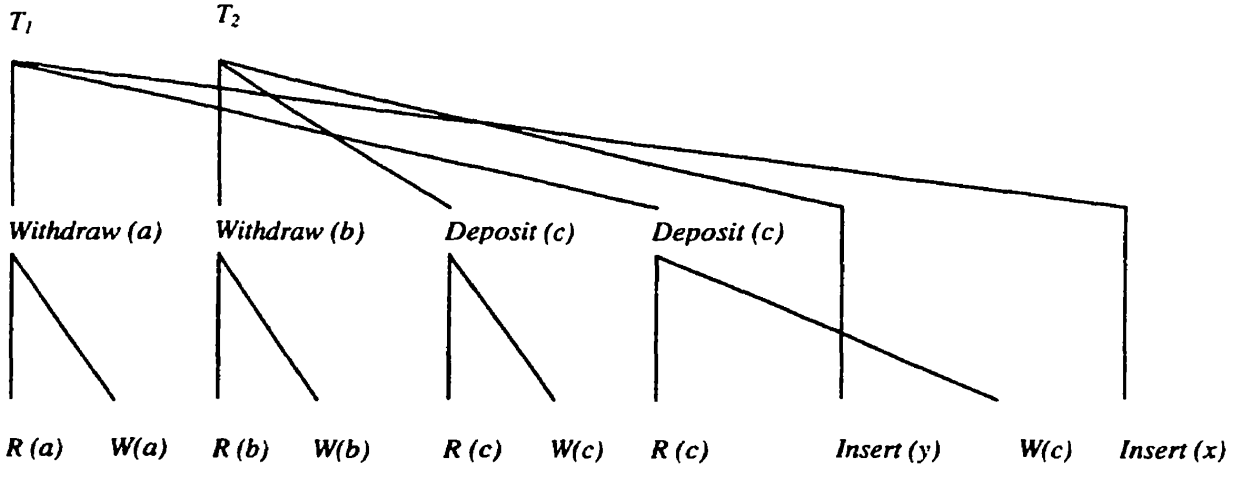

Figure 2.6 Concurrent Executions of Open nested Transactions (Courtesy: [WS92])

An open nested transaction (see Figure 2.6) is a generalization of the multilevel transaction. The difference between the two is that the former allows the transaction tree to have different nesting depths.

The open nested transaction model is different from Moss' [M81][M85] nested transaction model because they make the partial results of subtransactions visible to other top-level transactions before their parents commit. Clearly, the open nested transaction model relaxes the ACID paradigm. The isolation property of the transactions is relaxed by exploiting the semantics of the operations and by specifying which transactions are "open" and which ones are "closed". In an open nested transaction model, atomicity is achieved by using compensating transactions. The persistence of a subtransaction in an open nested transaction is undone by invoking compensating subtransactions.

The open nested transaction model finds use in extensible databases, federated databases, OO databases, and in exploiting transactions in operating systems. An important application, however, is that of exploiting intra-transaction parallelism where, concurrency control and recovery aspects are applied to the subtransaction executing within a top-level transaction.

2.4.3 Sagas

Sagas [GS87] address the delay problems that occur during the execution of long-lived transactions (LLT). LLTs hold on to the database resources for relatively long periods thereby delaying the execution of shorter and more common transactions. This may be due to the transaction accessing a large number of database objects or they have lengthy computations, or both. Examples of such transactions include transactions that produce monthly account statements in banks, transactions that process claims at an insurance company, *etc.* The other problem caused by LLTs is the increase in the transaction abort rate. Deadlocks occur due to the size of the LLT and the number of objects it accesses. These deadlocks eventually result in the abortion of transactions.

Sagas refer to a LLT that can be broken into a collection of subtransactions that can be interleaved in any way with other transactions. Each such subtransaction is a real transaction and hence they preserve database consistency. However, all the transactions in a saga are related to each other and must be executed as a (non-atomic) unit. The DBMS guarantees that either all the transactions in a saga are successfully completed, or compensating transactions are executed to amend a partial execution. Partial executions are undesirable and they must be undone. This type of processing allows a smaller unit of granularity. That is, whenever a portion of the transaction (subtransaction) is completed, the resources held by it are released. This significantly increases the concurrency in the case of lock-based concurrency control algorithms.

Sagas require compensating transactions to support recovery mechanisms due to their open nature. For each subtransaction in a saga, there must be a compensating subtransaction. They support *forward recovery* (aborting) and *backward recovery* (compensating) mechanisms. The isolation property is violated due to the revelation of partial results. The arbitrary interleaving of the subtransactions can sometimes violate the consistency property.

When sagas are applied to MDBSs, local autonomy is not severely violated because each element database sees each subtransaction as a local transaction managed by the element DBMS. Providing compensating transactions is a major difficulty in the case of sagas. Hence, sagas are useful in compensatable environments. They cannot be applied in scenarios where a transaction is irreversible, such as drilling holes. Nevertheless, sagas are appealing to compensatable MDB environments because they have minimal effects on the autonomy of the element databases.

Garcia-Molina *et al.* [GGK+91] provides a generalization of sagas called the *nested sagas*. For example, the activities in a data processing application can be implemented as nested sagas. In such applications, each subtransaction (saga) is treated as an independent activity that is further divided into its own sequence of steps and compensations. Any step 'X' in such an activity "thinks" that each of its subactivities are a collection of steps.

Further, an activity at a higher level may "think" that the activity 'X' is composed of several steps. Hence, aborts are propagated both up and down the tree of nested sagas.

2.4.4 Flex Transaction Model

The Flex transaction model [ELL+90] specifically addresses the transaction management issues in a MDB environment. It identifies the challenges posed by the autonomy of the underlying system and provides an extended transaction model with the following features:

- allows composition of *flexible transactions*,
- supports the concept of *mixed transactions*, and
- incorporates the *temporal aspects* of transaction processing.

Flexible transactions are based on the concept that a global transaction can be frequently completed successfully in more than one way. This implies that the global transaction is decomposed into a set of functionally equivalent subtransactions. For example, in a tour booking application, two transactions that book an air ticket on two different carriers to the same destination are said to be functionally equivalent. The global air ticket transaction can have different subtransactions that can accomplish the same task in different ways. The processing of the transaction continues even if one of the alternatives (subtransactions) fails. Such composition of transactions is called *flexible transactions*.

It is not necessary that all the subtransactions execute completely for the correct completion of the global transaction. This implies that atomicity at the global level is violated. However, the global subtransactions execute in an atomic fashion. The specification of the transaction execution alternatives implies the specification of the violation of atomicity. This specification must fit into the execution dependency existing among the subtransactions. These dependencies determine the legal execution order of the subtransactions and hence need to be specified when specifying a global transaction. Two types of dependencies exist – *positive* and *negative dependency*. These dependencies are actually the global integrity constraints used to maintain global consistency.

A positive dependency exists between two subtransactions t_1 and t_2, if t_1 waits for the results of t_2 before it starts.

A negative dependency exists between two subtransactions t_1 and t_2 if t_1 waits for t_2 to execute and fail. This is useful in cases where the results of t_2 are preferred over the results of t_1.

Mixed transactions are a combination of *compensatable* and *non-compensatable transactions.* A compensatable transaction is one for which a corresponding transaction can be specified which semantically undoes the effects of the committed transaction. This results in the violation of the isolation property. However, this results in enhanced concurrency, because this concept allows the global transaction to reveal its partial results to other transactions before it commits. A non-compensatable transaction is one for which a compensating transaction cannot be specified. For instance, a transaction that drills a hole or fires a missile cannot be undone after it commits. Flex transaction model allows the processing of both compensatable (open nested transactions) and non-compensatable (traditional flat transactions) transactions. In other words, it allows the processing of *mixed transactions.*

Due to the autonomous nature of the underlying system, the local database management systems decide on when to submit the global subtransactions. For instance, a bank transaction involving two banks in two different time zones could be processed at two different times [ELL+90]. In such cases, the temporal aspects of transaction processing must be taken into consideration. Flex transaction model does exactly that by associating a *temporal predicate* with each subtransaction. The temporal predicate indicates the time when the subtransaction should be executed. Other than this, the MDB environment has another temporal aspect to it – *transaction completion time.* This is the time within which a particular transaction must be completed. Flex transaction model uses these aspects of transaction processing in implementing the transaction scheduling mechanism.

Apart from its application in MDB environments, Flex is also applicable in CAD/CAM and CASE databases.

2.4.5 ConTract Transaction Model

Reuter [R89] describes a model for managing long-lived complex transactions in traditional transaction processing systems. A global transaction is divided into subtransactions (a sequence of steps) that are capable of defining how control must flow among themselves. Forward recovery and backward recovery mechanisms are suggested. Forward recovery suggests that the state information of all the transactions must be maintained. A compensating mechanism is required to support backward recovery. Due to its open nested structure the problems of sagas express themselves in this model as well.

The ConTract model is unsuitable in a MDB environment because it affects local autonomy to a large extent. The model is suitable only to those MDB environments in which:

1) all participating DBMSs can save the state information, and
2) the global transactions can be decomposed to global subtransactions and only a single global subtransaction is required at each participating element database, and whose visible two-phase commit can communicate the state information back to the MDBMS to ensure recovery.

Both these environments may eventually be realized, but they will not be true of all foreseeable MDBSs [BE99].

2.4.6 *ACTA*

Chrysanthis and Ramamritham [CR90] [CR91] [CR94] propose *ACTA* as a comprehensive transaction framework that facilitates the formal description of properties of extended transaction models. The need for a framework was realized due to the lack of functionality and efficiency of traditional models in complex applications. Examples of

such complex applications include, CAD/CAM, software development environments, object-oriented databases, stock trading databases, *etc.* Efficiency refers to the throughput demands placed on these systems, while functionality refers to the applicability of certain transactions in certain environments. For instance, the traditional transaction models were developed for short-lived transactions executing in competitive environments, while current applications require long-lived, interactive transactions running in collaborative environments. The simplest form of complex transactions executing in complex applications are Moss' nested transactions [M81][M85].

The semantics of transaction interactions are expressed in terms of transactions' effects on the commit and abort of other transactions and on objects' state and concurrency status (See Figure 2.7).

The *ACTA* framework also allows for specifying the *structure* and the *behavior* of transactions as well as for reasoning about the concurrency and recovery properties of transactions. The structure of the transaction refers to the nesting structure of a transaction, and the behavior refers to the operations invoked by a transaction.

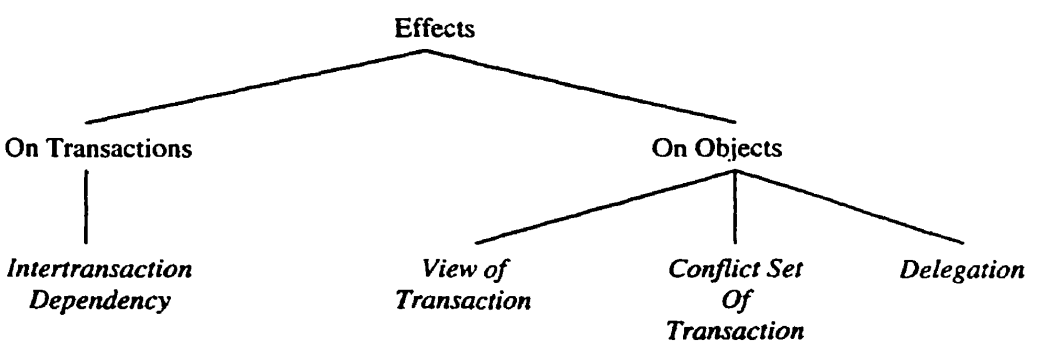

Figure 2.7 Dimensions of the ACTA framework (Courtesy: [CR94])

The behavior of a transaction processing system is determined by the behavior of the transactions executing in it and the objects manipulated by these transactions. In *ACTA*, the interactions among the transactions are expressed in terms of the transactions' effects on other transactions and the transactions' effects on the objects they access.

The effects of a transaction on other transactions are captured using the dependencies that exist among these transactions. There are two possible dependencies – *commit* and *abort*, collectively known as *completion dependencies*. A complete treatment of dependencies is available in [CR94].

A *commit dependency* between two transactions, *A* and *B*, indicates that *A* cannot commit until *B* either commits or aborts. The reverse may not be always true.

An *abort dependency* between two transactions, *A* and *B*, indicates that *A* must abort if *B* aborts. However, it does not imply that *A* must commit if *B* commits and *B* must abort if *A* aborts.

Transaction effects on objects are captured by the introduction of a *View Set* and an *Access Set*, and by the concept of *delegation*. Each object is characterized by its *state* and *status*. The *state* of the object is represented by its contents. This changes when a transaction accesses the object and modifies its contents. The *status* of an object is represented by the synchronization information associated with the object. It changes when a transaction performs an operation on the object. These concepts affect the visibility and other ACID properties.

Every transaction is associated with a set of objects that contains all the objects potentially accessible to the transaction. This set is called the *View Set*. This restricts the effects of the transactions on objects.

The objects already accessed by the transaction are contained in another set, called the *Access Set*. The objects accessed in the View Set become the members of the Access Set. These objects continue to be accessible to the transaction. The objects in the View Set are accessed by a transaction based on the concurrency control status of the object.

A transaction may delegate responsibility for finalizing its effects on some of the objects in its Access Set to another transaction. This is done by removing the objects from the Access Set of the first transaction (*delegator*) and adding them into the Access Set of the second transaction (*delegatee*). This process is called *delegation*.

Finally, the *ACTA* formalism can be used to show the correctness of a particular specification of a transaction model.

2.5 Leading Open Questions

Sections 2.3 and 2.4 presented several research efforts in the area of transaction management, concurrency control and recovery in multidatabase systems. Each generation of research provides solutions that address several issues of transaction management. The first-generation of research primarily dealt with systems classified as 'traditional transaction processing systems'. The solutions provided were suitable to those environments that are ideal where there are no performance requirements or failures occurring. However, such ideal systems rarely exist. Hence, the application of the results from the first-generation research finds little use when applied to specific domains. However, those results laid the foundation for further research and are still considered to be important.

The second-generation realized the need for more efficient methods of transaction management. This gave raise to few dimensions of research where the conservative approaches are enhanced by either relaxing the constraints in the environment or modifying the environment itself. For example, most researchers felt the need to relax the correctness criteria for concurrency control and ACID properties of transactions. These efforts paved the way to several results that attempted to relax global serializability. The highlight however was finding that semantic knowledge of transactions aids in enhanced transaction management. However, the problem of providing semantic information to transaction processing system still exists.

The third-generation of research realized the necessity of extending the existing transaction models so that they can fit a particular problem domain. Several transaction models were proposed, each based on a specific application domain. Each model has its own concurrency control and recovery methods. Almost all transaction models are broadly based on the nested transaction model [R78][M81][M85]. The application of semantic knowledge in such transaction models helps in leveraging the efficiency of such models. However, the specification of semantics still remains a problem. Most of these transaction models have been developed for traditional transaction processing systems or distributed database systems. The application of such systems to a MDBS is an interesting research area. Nested transaction model provides interesting results when applied to an MDBS. Sagas could be applied to MDB environments, but only to those that are compensatable. This is also the case for most of the transaction models that have a flavor of open nested-ness in them. In compensatable environments, providing compensating transactions to support backward recovery is a difficult task. We do not use sagas in our environment because of the nature of the transactions executing in our environment. Sagas are attractive if the transactions are primarily long-lived and compensatable as is the ConTract model. ConTract is not a solution to the MDB environment considered in this thesis because it violates the autonomy of the element databases. Flex transactions [ELL+90] are very interesting to this thesis. Our goal of providing multiple (transaction) execution alternatives is similar to their approach. However, as we explain our paradigm, the differences between the models become evident (see Chapters 3 and 4).

With the knowledge of several research efforts in the area of transaction management in MDBS, we identify the following problems:

1) Development of transaction models suitable to address the transaction management issues in a MDBS that is used as a back-end in an Internet environment for specific application domains, and then generalizing the same,
2) Automatic generation of the semantic information of the transaction (specifically) and the application (broadly),

3) Exploiting transactional dependencies to enhance the concurrent execution of transactions,
4) Development of a correctness criterion for concurrency control in such transaction models,
5) Modifying the MDB environment at the operational level to support the transaction model and associated concurrency and recovery methods,
6) Characterizing the local autonomy interface, and
7) Development of standards for transaction processing in Internet environment.

2.6 Summary

This chapter started with the discussion on the multidatabase architecture and transaction management problem. Section 2.3 discussed the various approaches to the transaction management problem. The discussion covered the first- and second-generation research efforts. Section 2.4 provided the third-generation research efforts that attempt to develop application-based transaction models. Based on the background material and related work discussed in Sections 2.3 and 2.4, Section 2.5 outlines the various leading research problems in the area of transaction management.

We observe that the past research efforts do not address the entire range of issues identified. We use the material discussed in this chapter as a platform for our work. Chapter 3 discusses the transaction execution dependencies identified in our model followed by a discussion of the results from our experiments with it.

The significant problems we face cannot be solved at the same level of thinking we were at when we created them.

- Albert Einstein

Chapter 3

Transaction Model and Execution Dependencies

This chapter describes our transaction model and transaction execution dependencies. Although Ehikioya and Barker [EB97] provide a formal treatment of execution dependencies using the concept of causality, it is more mathematical than how we treat them here. We start with the introduction and description of the model followed by a discussion on execution dependencies. We formally define the execution dependencies followed by a discussion describing how to exploit them within the framework of our transaction model to enhance intra-transaction parallelism and provide multiple transaction execution alternatives.

The research goal is to develop a transaction model and identify the execution dependencies in it to provide enhanced intra-transaction parallelism thereby producing multiple transaction execution alternatives. The requirement for an advanced transaction model exists for all the reasons explained in Chapter 2. Intra-transaction parallelism is of high importance in any transaction processing environment that uses any form of nested transaction model. Most business applications can be mapped into a multidatabase architecture at the data source level. This illustrates the autonomy problem inherent in

such systems. Care must be taken to preserve the autonomy of such systems. Any application running on such systems must be developed so it does not violate the autonomy at the element databases.

Present day applications involving transaction processing require the system to be tolerant to subtransaction failures. That is, the MDBS transaction must not fail completely just due to the failure of a part (subtransaction) of it. This requirement is experienced in many environments that provide a choice for achieving a general "global" goal. For instance, an air ticket reservation system helps a travel agent book the same ticket on multiple carriers. This implies that the system uses multiple transactions, all with the same objective, that is, to book an air ticket to the same destination.

Section 3.1 introduces our transaction model. In Section 3.2 we discuss execution dependencies. A discussion of exploiting the execution dependencies within the framework of our transaction model to enhance intra-transaction parallelism is presented in Section 3.3. It also discusses multiple execution alternatives. This chapter is once again concluded with a summary.

3.1 Transaction Model: Description

Chapter 2 introduced the reasons why advanced transaction models are needed. Keeping those reasons in perspective, we develop a transaction model (see Figure 3.1) suitable to serve a certain domain of applications. Recall the primary requirements of an advanced transaction model from Chapter 2 (Page 33).

An advanced transaction model must provide better support for long-lived activities. It must also provide mechanisms to relax the ACID paradigm thus helping to capture the semantics of the operations. These requirements also yield (inter-/intra-) transaction parallelism. Further, they provide support to transactions executing in a federated database environment.

In addition to the above reasons, present day applications require multiple alternatives for transaction execution to minimize the effects of subtransaction failures. We attempt to relax the ACID paradigm of transactions to implement an open nested transaction model in a MDB environment. Further, at an operational level, we realize a multi-layered architecture. This becomes evident as we describe our model.

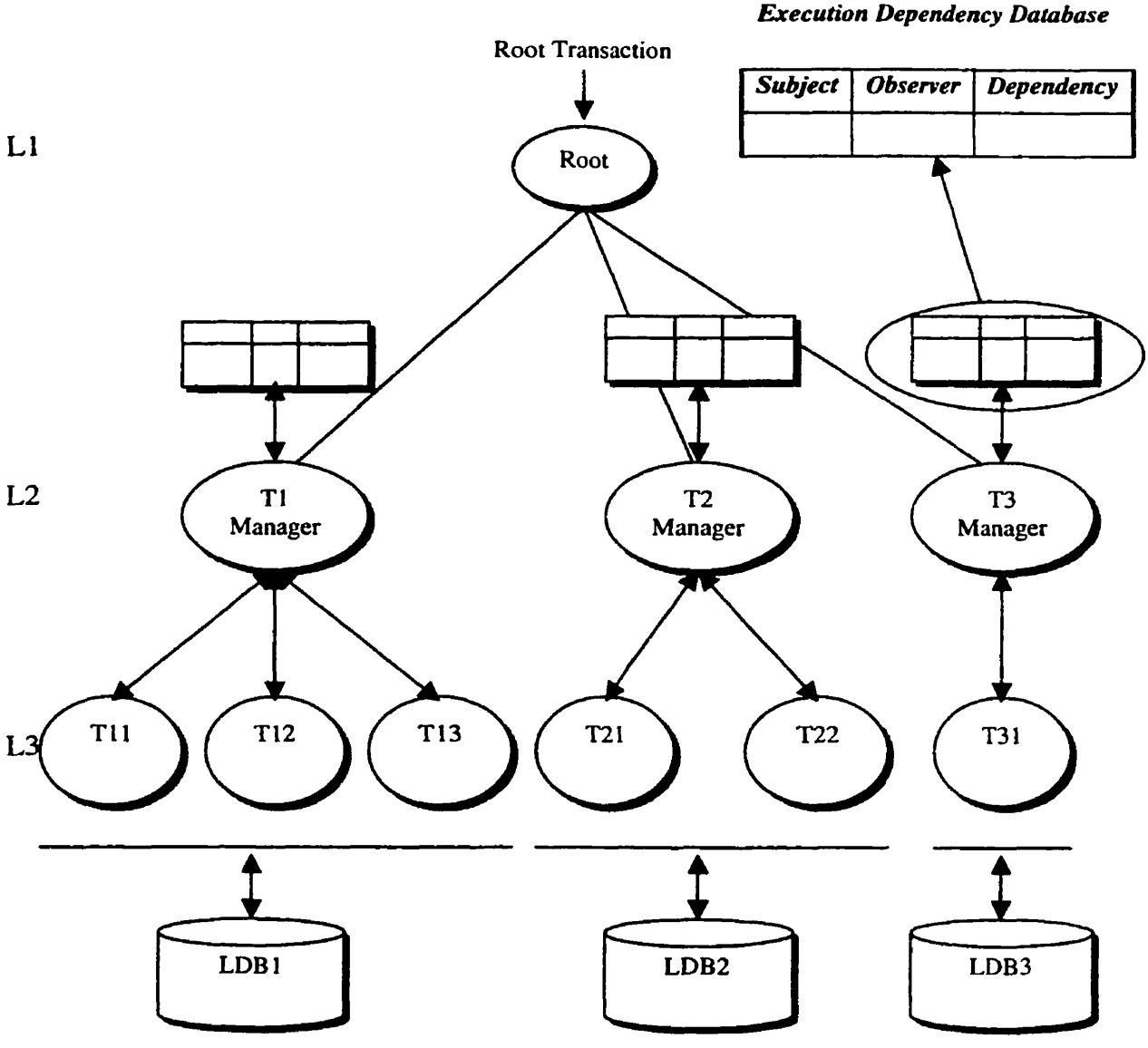

Figure 3.1 Transaction Model Architecture

Our transaction model is a nested transaction one defined over a MDB environment. The subtransactions reveal their partial results and hence we have an open nested transaction

model. While this model uses an MDB environment as the underlying source, its implementation uses layered architecture of databases and subtransaction managers to characterize the autonomous interface to the underlying systems (see Figure 3.1 on Page 50).

Figure 3.1 shows a generalized nested transaction model of nesting depth, $n=3$. The root transaction has three subtransactions each of which has subtransactions. The subtransactions at L_2 also act as managers denoted with the name of the transaction, suffixed by *'Manager'*. For example, the manager at a subtransaction arbitrarily named *T2* is denoted as *'T2Manager'*. Each subtransaction is an object with two roles – *'subtransaction'* and *'subtransaction manager'*. The managers exist just below the root level though they can exist anywhere between there and just above the leaf transactions. For example, suppose a nested transaction has *'n'* levels. The levels are numbered so the smallest ordinal is the root and the largest ordinal is the leaf. The transaction would appear as shown in Figure 3.2.

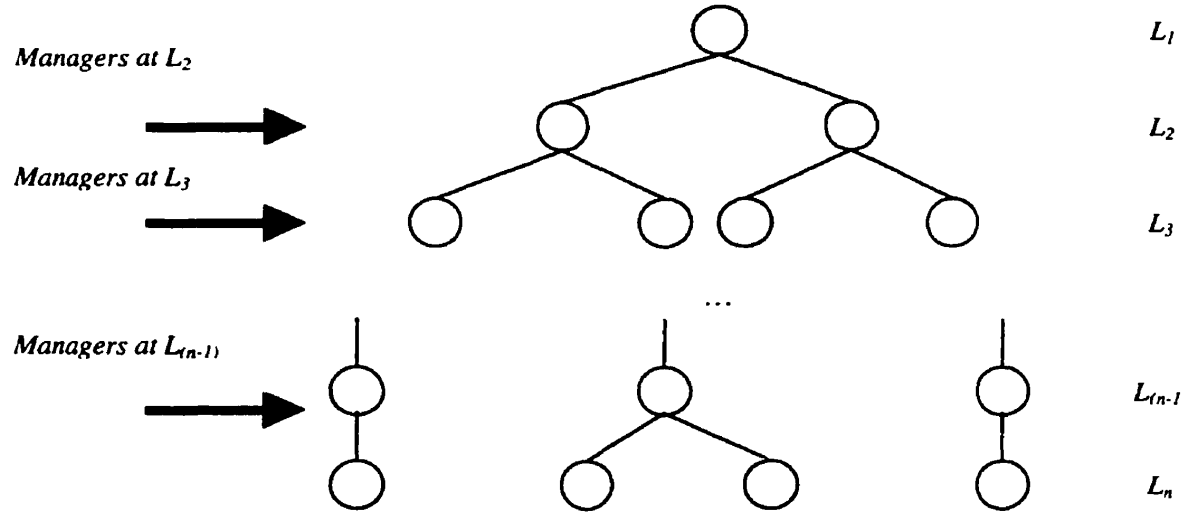

Figure 3.2 Example Nesting Illustrating Positioning of Subtransaction Managers

The subtransaction managers could be anywhere between L_2 and $L_{(n-1)}$ inclusive, as shown using dark arrows in Figure 3.2. The rationale behind placing subtransaction managers at those levels is to maintain the autonomy of the underlying system as much as

possible. It is observed that if the managers are at a lower level, closer to the leaf transactions, then the amount of autonomy violation is more than when the manager is at a higher level, closer to the root transaction. Each subtransaction manager has a database of execution dependencies that its subtransactions have with others within a MDBS transaction. For instance, consider the managers shown in Figure 3.1 on Page 50. It can be seen that the managers are at L_2. These could have been positioned at any of the other levels in the transaction model too. However, it can be observed that if the managers were at a level lower than the current level, then the autonomy of the databases is violated to a larger extent than when they are above the current level. The reason for this violation is the amount of communications between the managers at the underlying systems in the former case than their management systems in the latter. Execution dependencies are discussed in Section 3.2.

The database at the subtransaction managers contains information about the subtransactions that are (execution) dependent and the type (strength) of dependency between them. After a subtransaction produces a partial result, it notifies its parent (subtransaction manager) about the same. The subtransaction manager looks into the execution dependency database to check for the dependencies its subtransaction has with other subtransactions. Based on the dependency it shares with other subtransactions, the corresponding subtransaction managers are notified of the partial result. On receiving the notification, the subtransaction managers at the receiving end spawn their subtransactions. Meanwhile the notifying subtransaction may have executed all its subtransactions (if any) to completion and the availability of all those results are also notified to other subtransactions based on the information in the execution dependency database. This process repeats until the root transaction's objective ("global" goal) is accomplished semantically.

Letting other subtransactions know of partial results violates the isolation property of a transaction. Conversely, the violation of the isolation property implies the use of the principles of an open nested transaction model. However, the intra-transaction parallelism is enhanced for the same reason. The concept of managers yields multi-layered database

architecture above the MDB environment. The execution dependency database contains the information about the subtransactions participating in the dependency relationship and the type (strength) of dependency. This allows one or more subtransactions to observe one or more other subtransactions until the latter produces a necessary (partial) result. The former set of subtransactions is called *observers* while the latter set of subtransactions is called *subjects*. The notification is sent to the observers based on the type of dependency existing between them. These dependencies (See Section 3.2) are the global integrity constraints and are implemented only at the subtransaction levels.

Definition 3.1

Observers: A set of subtransactions observing another set of subtransactions whose (partial) results could be of potential use to execute its own atomic operations or their subtransactions. ■

Definition 3.2

Subjects: A set of subtransactions observed by another set of subtransactions so the formers (partial) results have potential use to the latter to execute its own atomic operations or their subtransactions. ■

For example, given a MDBS transaction *T*, with 4 subtransactions *ST1*, *ST2*, *ST3*, and *ST4* contributing towards achieving a "global" objective. If *ST1* and *ST3* are (execution) *dependent* on the partial results of *ST2* and *ST4*, then:

1) *Observers = {ST1, ST3}*
2) *Subjects = {ST2, ST4}*

Note that the observers may also behave as subjects and vice versa in the context of other subtransactions based on application semantics. The violation of autonomy is minimal when the constraints are closer to the root level and more when they are closer to the leaf level. Hence, based on the specification of the global transaction, different transaction executions can be realized with varying degrees of autonomy violation. It is observed that

the autonomy violation increases as the subtransaction managers are moved toward the leaf transactions. The positioning of the subtransaction manager defines the interface that characterizes the autonomy of the underlying MDB environment.

Our transaction model is broadly based on the open nested transaction model. However, at an operational level we observe multi-layered architecture comprising execution dependency databases and subtransaction managers over the MDB environment. The concept of subtransaction managers characterizes the violation of autonomy of the MDBS architecture. Partial results are exposed due to the openness of the transaction model. This results in the violation of the isolation property thereby relaxing the ACID model.

3.2 Execution Dependencies

Advanced transaction models are designed to cater to specific application requirements in a distributed database system. When these models are applied to MDB environments, the characteristics of the MDBS add to the complexity of the transaction management. Specifically, the autonomy of the element databases affects the execution of a MDBS transaction. Execution dependencies extend the semantics of the transaction model to enhance intra-transaction parallelism thereby providing multiple execution alternatives when a part of the MDBS transaction fails. This is similar to the flexible transactions in Elmagarmid's Flex transaction model [ELL+90]. The transaction in the Flex model is a two-level nested transaction whereas in our model the transaction could have an arbitrary number of nesting levels. However, the similarity in these approaches is the provision of multiple execution alternatives using functionally equivalent subtransactions, and maintaining a high degree of autonomy at the underlying systems.

Certain applications require extensive use of semantics to ensure the successful completion of the MDBS transaction. The reasons are:

1) to enhance parallelism within a transaction,

2) the application may require the use of multiple choices of execution to achieve a global objective, and
3) the subtransactions may fail to produce a certain expected result (for example, an air ticket reservation in a particular airline).

The above reasons are the motivations for the development of our transaction model. Further, these motivate us to find the execution dependencies existing among the subtransactions to enable us to enhance the intra-transaction parallelism. We are also required to categorize the dependencies based on the application requirement. At the same time, the MDBS characteristics require us to maintain the degree of autonomy at the element databases at the highest level. All these reasons lead us to characterize execution dependencies into three types: *F*, *N* and *B* as defined in Definition 3.3.

The *'F'* dependency takes the highest priority over the *'N'* and *'B'* dependencies if they all exist among the same set of observers and subjects. In the absence of *'F'* dependency, *'N'* dependency is superior to the *'B'* dependency. They are useful when they provide results that add more meaning to the semantics of the MDBS transaction. One example is when an economy air ticket is upgraded to a business class ticket in an air ticket transaction. This does not change the semantics of the air ticket transaction. The failure of the transactions that have a *'B'* dependency also does not affect the semantics of the MDBS transaction. For example, suppose the booking of a window seat in a preferred carrier fails but there is some other seat available in the same carrier. The air ticket transaction's semantics of booking a ticket to a particular destination on that particular carrier are still valid.

Definition 3.3:

Execution Dependency: An *execution dependency (ED)* exists between a *subject* subtransaction and an *observer* subtransaction of a global transaction based on a dependency that is *sufficient*, *necessary*, or *bonus*. Thus *ED* is defined as a triple as follows:

$$ED = \{S, O, D\}$$

where,

S is a subject subtransaction,

O is an observer subtransaction, and,

D is a dependency from the set *{F, N, B}*

where,

F is sufficient,

N is necessary,

B is bonus. ■

The *sufficient dependency (F)* between subject and observer subtransactions indicates that the (partial) results of the former is just sufficient to trigger the latter's execution. However, the *necessary dependency (N)* between subject and observer subtransactions indicates that the (partial) results of the former is necessary to trigger the latter's execution. The *bonus dependency (B)* is utilized in enhancing the semantics of the application. However, as mentioned earlier, the sufficient dependency takes priority over the necessary and bonus dependencies in case of concurrent executions to enhance the intra-transaction parallelism. Similarly, the necessary dependency takes priority over the bonus dependency when both exist between the same set of subject and observer subtransactions.

The execution dependency information is available in the execution dependency database at the subtransaction managers of our transaction model. This information is used for two purposes:

1) to enhance the intra-transaction parallelism in an MDBS transaction through the extension of transaction semantics, and
2) to provide multiple execution alternatives to achieve the global objective of a MDBS transaction.

Depending on the type (strength) of dependency, the MDBS transaction execution flows differently. Multiple execution alternatives are automatically realized in this model. Hence a MDBS transaction can be successfully completed through the execution of various sets of its subtransactions with enhanced intra-transaction parallelism.

Example 3.1

Consider an online client booking an accommodation as part of his trip. He might want to reserve accommodation only if he has an air ticket. Hence the constraint for the execution of the accommodation transaction is the successful completion of the air ticket transaction. This implies an execution dependency between air ticket transaction (subject) and accommodation transaction (observer). For this example, let us assume that the air ticket transaction executed successfully. There are different types of accommodation the client could request. For instance, he could reserve a hotel, motel or hostel. Based on the clients' requirements, there are multiple transaction execution alternatives available. That is, he could either book a hostel or a motel if the hotel subtransaction fails.

The client spawns the trip transaction. The accommodation and air ticket transactions are the subtransactions of the trip transaction. According to the semantics of the trip transaction, the execution of the accommodation subtransaction depends on the successful completion of the air ticket subtransaction. On successful completion of the air ticket subtransaction, it notifies the accommodation subtransaction of the same. At this point the accommodation subtransaction spawns its subtransactions (hotel, motel and hostel subtransactions). If the hotel subtransaction fails, and the functionally equivalent motel or hostel subtransaction returns a positive result, then the trip is booked based on what is available (provided the client is also happy with the booking!). ■

The execution dependencies are present in the subtransaction manager's lookup database. All the subtransactions look up their execution dependency database after they obtain a partial result from their children. That is, a subtransaction that has a partial result notifies the dependent subtransactions based on the information in the lookup database at their manager. If all three dependencies exist between the same observer and different

subjects, the subjects first notify the observers with which they have an *'F'* dependency. The observer starts to execute just after it receives the notification from the subjects' managers. An *'F'* dependency is given priority to decrease the wait time of the observer subtransaction. This directly enhances the intra-transaction parallelism in a MDBS transaction. Each observer subtransaction is either subdivided, or is a flat transaction. All the subtransactions of the observers provide multiple transaction execution alternatives. That is, any execution of the observers' subtransactions yields results that satisfy a common goal (maintains the semantics) of their parent transaction. For instance, an air ticket subtransaction could have subtransactions; one each for an economy ticket and a first class ticket. Both these subtransactions provide a common goal of booking a ticket to the same destination (this is specific to the application).

Identifying these dependencies aid us in two ways as is evident from the preceding discussion:

1) To use them in enhancing the intra-transaction parallelism, and
2) To provide multiple execution alternatives by extending the semantics of the application.

The exploitation of semantics and the dependencies to enhance intra-transaction parallelism affects the ACID model and the autonomy of the element databases. However through the use of subtransaction managers, we maintain a high degree of autonomy. This, in fact, can be tuned based on the needs by shifting the subtransaction managers either upward in the hierarchy to achieve higher degree of autonomy (useful in MDBS) or downward in the hierarchy to achieve a lower degree of autonomy (useful in general distributed database systems).

3.3 Discussion

The execution dependencies are specified at the start of the MDBS transaction. These dependencies are stored in the appropriate subtransaction managers based on participating subtransactions. It is stored in the lookup database along with the observer

and subject subtransactions. The managers are present at the subtransaction levels only. The positioning of the subtransaction manager decides the characterization of the autonomy interface of the underlying MDBS architecture.

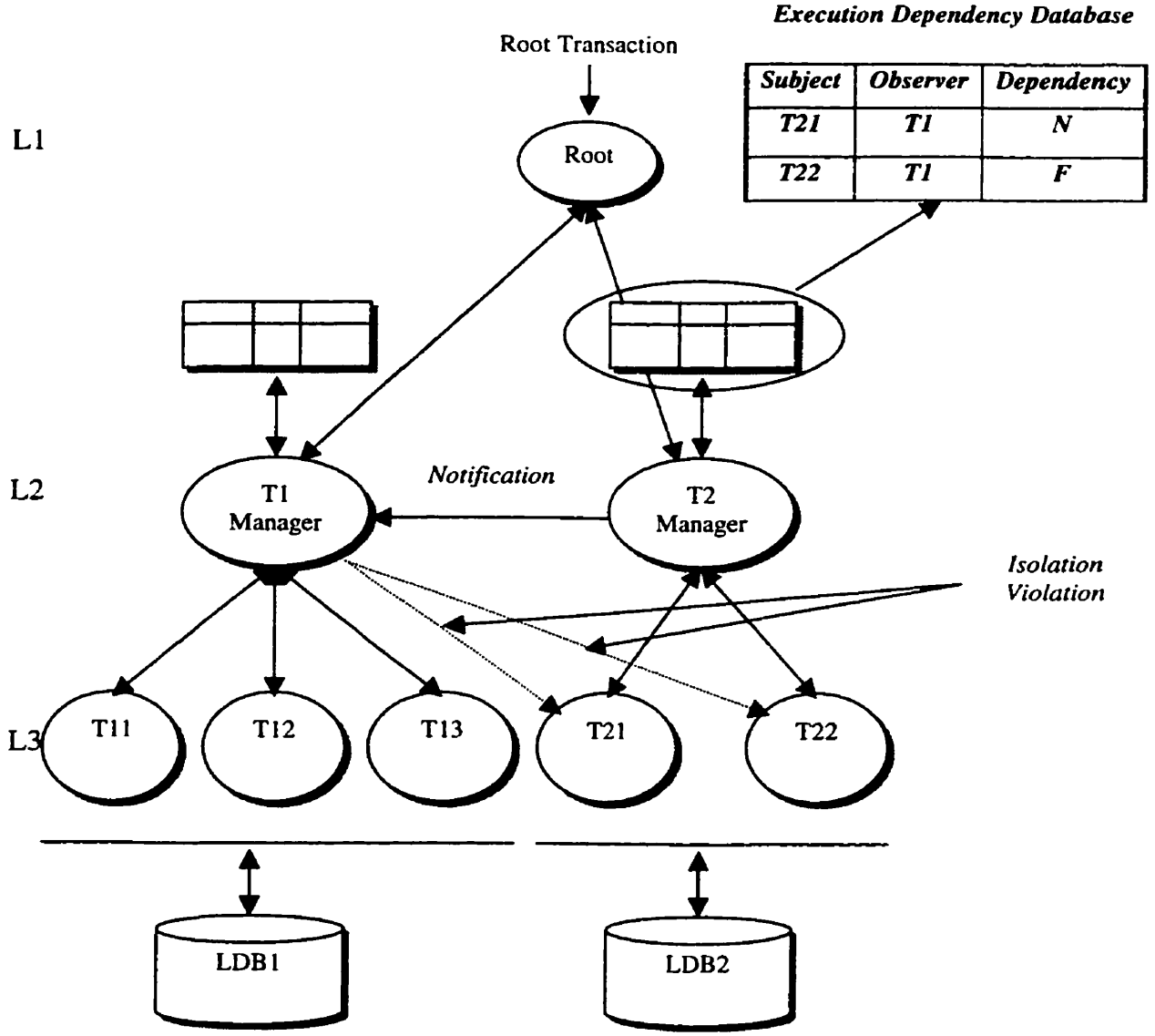

Figure 3.3 An Illustration of Execution Dependency

Consider the transaction illustrated in Figure 3.3. The root transaction has two subtransactions *T1* and *T2*. *T1* has three flat subtransactions *T11*, *T12* and *T13*. *T2* is divided into two subtransactions *T21* and *T22*. The subtransaction managers are present at level *L2* where the subtransactions *T1* and *T2* are present. The execution dependency

database is present at the subtransaction managers. Actually, each of these subtransactions plays two roles at the same time – *'subtransactions'* and *'subtransaction managers'*. The subtransaction managers use the execution dependency database. The execution dependency database has information about its subtransactions and the dependency it shares with other subtransactions in the hierarchy.

Figure 3.3 shows the execution dependency information available at *T2Manager*. It shows that subtransaction *T21* is the subject of observer *T1* with a dependency *'N'* and *T22* is the subject of observer *T1* with a dependency *'F'*. From the definitions of dependencies it can be said that *T1* executes if it receives the partial result (*T22*) from *T2Manager*. This is because an *'F'* dependency takes priority if there is such a dependency between the same observer and different subjects in the execution dependency database. However, in the absence of an *'F'* dependency, *'N'* dependency takes priority, if one exists. Suppose *T2* starts executing *T21* and *T22*. Once it receives results from *T22* before *T21* executes to its completion, the *T2Manager* looks up the execution dependency information and notifies *T1Manager* about the available partial result. This is because the dependency information specified states that *T22* results are just *sufficient* for *T1* to execute. Now *T1* spawns *T11*, *T12* and *T13* and waits for the results from either of these and notifies the corresponding subtransaction, or the root (in this case) about the available result. However, the *T21* results are also passed on to *T1Manager* by *T2Manager*, which by then would have started executing its subtransactions. This enhances the intra-transaction parallelism. Since the subtransaction managers are implemented closer to the root transaction, autonomy at the element databases is preserved. The concepts of subtransaction managers and that of subjects and observers are implemented using design pattern techniques [GHJ+95]. The notification mechanism has been implemented at the subtransaction manager level at which they notify the corresponding subtransaction managers depending on the information available in the execution dependency database (see Figure 3.3).

Gamma *et al.* [GHJ+95] define design patterns as descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context.

A design pattern names, abstracts, and identifies the key aspects of a common design structure that make it useful for creating reusable object-oriented design with each pattern focusing on a particular issue. This thesis presents a specific application of the design pattern techniques. This specific example could be generalized so that template code could be produced to assist in the design of other nested transaction applications. For example, the trip booking application is just an example to demonstrate the application of design pattern techniques. The template of this application could be extended to capture other nested applications, such as, auctioning systems, information kiosks, *etc.*

In our transaction model, the communicating objects are the observer and subject subtransactions. Their communication relaxes the ACID model by exploiting the openness of the nested transaction model. This, in effect, enhances the parallelism of subtransactions executing within the MDBS transaction. The nesting level at which the subtransaction managers are placed decides the autonomy of the underlying systems. It is observed that our model is a combination of the open nested transaction model (because the subtransactions reveal their partial results) and multilevel transaction model (because of the multi-layered database design realized at an operational level due to the presence of the execution dependency database at the subtransaction managers).

3.4 Summary

The chapter began with the introduction of our advanced transaction model. It discussed the use of subtransaction managers and the execution dependency databases. Section 3.2 presented the execution dependencies. We also presented the representation of the execution dependencies in the subtransaction managers. Section 3.3 presented a discussion about the use of execution dependencies within the framework of our transaction model. Chapter 4 discusses the use of execution dependencies and exemplifies the new transaction paradigm using a sample application.

An invasion of armies can be resisted, but not an idea whose time has come.

- *Victor Hugo*

Chapter 4
An Application of New Paradigm

This chapter starts with an analysis of the transaction management problem in multidatabases from the perspective of application semantics. Section 4.1 presents the problem analysis. In Section 4.2 we present an overview of our transaction model. Section 4.3 introduces an example to illustrate the application of the new paradigm. The illustration explains how our transaction model is utilized in executing the transactions of the application. Further, it explains how multiple transaction execution alternatives are facilitated. Section 4.4 presents a comparison of the new paradigm with the conventional models. The chapter details the observations made, and provides insights about this methodology, before it ends with a summary.

4.1 Problem Analysis

This section analyzes the transaction management problem from the perspective of application semantics.

As mentioned in earlier chapters, the transaction management problem in a MDBS is a critical and challenging problem. The objective is to address the concurrency and

reliability issues in transaction execution. This is coupled with maintaining autonomy among the heterogeneous databases that participate in the MDBS federation. This thesis attempts to enhance the intra-transaction parallelism available in an open nested transaction environment. The open nested transaction executes in a MDB environment. This thesis characterizes the autonomy among several databases at the source level. We exploit the application semantics to identify the dependencies among the several subtransactions in the nested transaction. This thesis also presents a transaction execution framework that provides multiple transaction execution alternatives.

A nested transaction is one where the objective of a transaction is achieved in steps (subtransactions) thereby maintaining parallelism and localizing potential failures. However, in such transactions the ACID properties are maintained just as in a flat transaction. In open nested transaction, the ACID properties are relaxed because partial results of the subtransactions are revealed. This enables the system to exploit the application semantics to enhance the intra-transaction parallelism. In this thesis, since our focus is on the concurrency issues, we do not discuss the reliability issues. Interested readers are referred to Chapter 2, which includes a discussion on the reliability aspect of transaction management. The exploitation of the semantic information of the application and/or their transactions opens interesting opportunities to address the transaction management problem.

A nested transaction submitted to the MDBMS is divided into several subtransactions. These subtransactions are submitted to the various LDBMS in the MDBS. The databases are characterized by autonomy at the data source level. This implies that the subtransactions cannot reveal their partial results to other subtransactions. Introducing subtransaction managers circumvents this problem. They are responsible for communicating the necessary results to other subtransactions. Hence, this requires an implementation of an interface that characterizes the level of autonomy the system offers. The concept of subtransaction managers is explained in the context of an application in the following sections.

The implementation described in this dissertation assumes the underlying systems to be homogeneous. However, it can be observed that the solution can be extended to a heterogeneous environment as is discussed in the future work section of Chapter 5.

4.2 Our Transaction Model

An advanced transaction model was presented earlier. This model is broadly based on Moss' [M81] [M85] nested transaction model and Weikum's [W90] multilevel transaction model. The purpose of this model is to serve advanced applications where concurrent transactions are common. Our model uses an open nested transaction with layered database architecture at an operational level. The underlying system of this model is a multidatabase environment. The transaction model characterizes the autonomy of the underlying MDBS through an interface, made of subtransaction managers and execution dependency databases, that can be moved either up or down in the layered architecture of the transaction model.

The transactions in the paradigm presented here are open nested and hence can reveal their partial results. Since intra-transactional concurrency is an issue addressed by this thesis, we develop a mechanism by which the transactions communicate with each other to reveal their partial results. The concept underlying the communication between the transactions is broadly based on behavioral design patterns [GHJ+95]. The communication between the transactions is based on the (execution) dependency dictated by the application semantics.

The concepts of transaction processing expressed by this thesis can be generalized to various application environments. The transaction model presented here finds use in several business environments that can be mapped into multidatabase environments in which the underlying systems require a high degree of autonomy. Examples of such application environments include trip planning, auctioning web sites, information kiosks, *etc.*

The paradigm presented here utilizes the concept of nested transactions with subtransactions cooperating to achieve a global objective. The implementation of the model has components that communicate to relax the ACID requirements at the subtransaction levels. These components provide a framework that can be utilized to customize a particular application running in a MDB environment demanding a high degree of autonomy. The components have two roles namely, subjects or observers (see Definitions 3.1 and 3.2 on Page 53). Based on the role of the components, they communicate with other components using the concepts of behavioral design patterns. This process of communication involves the components subscribing to one or more components as observers. The application semantics and execution dependencies determine the execution of transactions. The transaction execution alternatives are determined based on the application semantics.

For example, in an auctioning web site, suppose a transaction is required to bid and buy a wine goblet used by King George V. The subtransactions of this global transaction in the context of our transaction model could be one that bids, another that buys and the last one that verifies the credit of the bidder before the item is auctioned. In the context of our transaction model, these subtransactions could be encapsulated into the components in the framework presented here. Based on the application semantics, the execution dependency database is populated at the subtransaction managers. The knowledge of the dependencies and the semantics of the transaction execute the transaction to achieve the global objective of buying the wine goblet. The subtransactions act as subjects and observers in the process of execution thereby enhancing the intra-transaction parallelism. Multiple transaction execution alternatives are also possible. For example, after the bidding, when the subtransaction checks for the credit of the bidder, it could possibly check several credit card databases and utilize the one the bidder prefers, or the one that has credit (depending on application semantics).

Another example would be a customer looking for a particular kind of mountain bike in an Information Kiosk. The processes of identifying the store that carries the bike, buying the bike, and verifying the credit of the customer before selling the bike could be

represented as a nested transaction in our framework. All the above processing could be represented as subtransactions. As mentioned earlier, multiple parallel transaction execution is possible in this case as well. For instance, the preferred bike could be available at various stores on the kiosk. The customer could be provided with various alternatives based on his requirements (depending on application semantics). Ehikioya and Walowetz [EW99] present a formal specification for such e-commerce applications.

The concept of maintaining the autonomy of the underlying systems is evident in the above applications. For example, if the credit checking subtransaction has subtransactions of its own, the nesting depth of the entire transaction increases. Now based on the autonomy requirement, the subtransaction managers of this particular application could be placed either closer to the root transaction or to the leaf transactions. The violation of autonomy is higher in the latter. Hence the subtransaction managers of the nested transaction act as an interface to maintain the autonomy of the underlying systems. The execution dependency databases are present at the subtransaction managers, which along with the managers, provide a layered architecture.

4.3 An Example Application: Trip Booking

An advanced transaction model provides better support for application-specific environments. It also provides mechanisms to relax the ACID paradigm thus helping to capture the semantics of the application. It enhances the parallelism of the execution of transactions. The goal of this thesis is to provide a framework to execute an open nested transaction, in a MDB environment. The framework must have provisions for exploiting the dependencies between the subtransactions. Based on the dependencies identified, it must provide multiple transaction execution alternatives. Further, the framework must have provisions for maintaining a high degree of autonomy at the underlying systems in the MDB environment. This section explains the concepts of our transaction model using an application with nested transaction execution as shown in Figure 4.1.

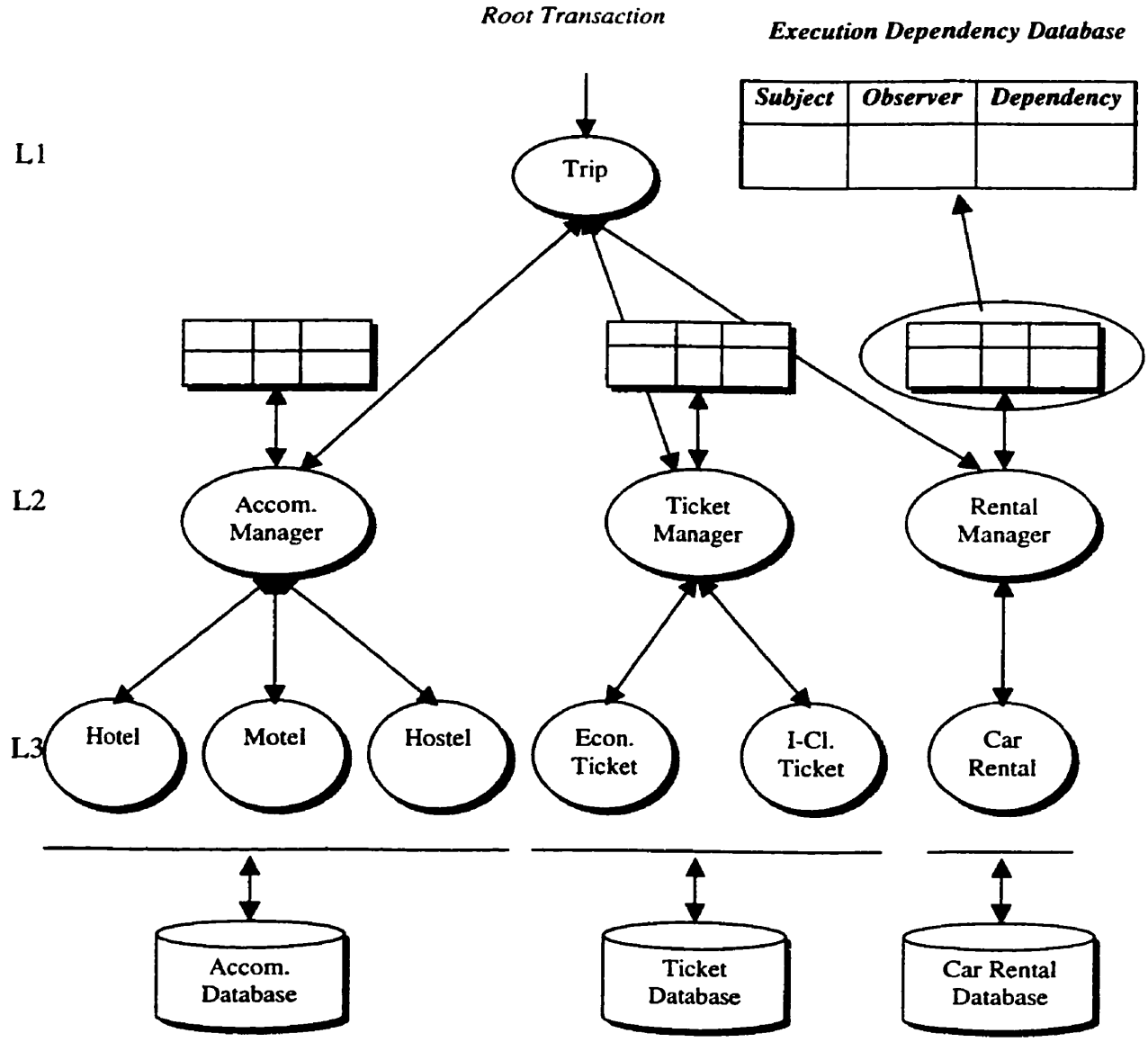

Figure 4.1 Transaction Model and Example Transaction Execution

A trip booking application [E92] is considered in this thesis to explain the concepts introduced in Chapter 3. A trip booking consists of at least three steps – booking air tickets, renting a car, and reserving an accommodation. The air ticket is either an economy or a first class ticket. Similarly, the accommodation is reserved either in a hotel, motel, or hostel. Our root transaction is the trip transaction with three subtransactions that book air tickets, car rental and accommodation, respectively. Further, these

subtransactions have subtransactions of their own. For instance, the air ticket subtransaction has subtransactions that book either an economy or a first class ticket.

The underlying environment of the trip booking application is an MDB environment. It has three autonomous databases that have information about air ticket, car rental and accommodation availabilities. Hence each subtransaction interacts with only the corresponding database. For instance, the accommodation subtransaction interacts only with the database that contains information about the hotel, motel and hostel. Figure 4.1 shows the nested transaction in our application and the different databases with which it interacts. The execution dependency database is shown at each nesting level. This figure is similar to the general one presented earlier in Chapter 3. A particular transaction execution scenario is presented here to explain the various communications that occur between the subtransactions. This example captures the various concepts described in this thesis.

The objective of the transaction in the trip booking application is to book a trip based on the customer's preferences. Further, it must enable multiple outcomes from the execution of the transaction. The transaction model *per se* captures the application semantics. The model enhances the intra-transaction parallelism by allowing the subtransactions to view the partial results of other subtransactions based on the captured application semantics.

In Figure 4.1 the different subtransactions at level L2 have been called subtransaction managers. The positioning of these managers is critical to the characterization of the autonomy interface. The higher the level at which these managers are placed, the more autonomous is the underlying system. Since the trip transaction has three subtransactions at L2, it can be observed that there are three managers. These managers are the subtransaction managers and may or may not have their own subtransactions. Each of these managers has a structure called an *execution dependency database* that stores dependency information among the subtransactions. Figure 4.1 also shows such a dependency database at the car rental manager. These databases and their managers provide a layered architecture to the transaction model at an operational level. The

dependency information is populated in these databases by capturing the application semantics. This happens when the transaction starts executing. The subtransaction managers play the roles of subjects and observers (defined in Chapter 3 – Definitions 3.1 and 3.2 on Page 53). Based on the outcome of the subject and the dependency it has with the observer, the subject communicates the information (i.e., reveals its results) to its observer(s). For instance, suppose the booking of an airline ticket is dependent on renting a car. The application semantics imply that there is a (execution) dependency between the rental and air ticket transaction. This information is stored in the execution dependency database at the car rental manager. The car rental transaction executes against the car rental database to find if there is a car rental as per the request. Based on the result of the transaction and the dependency information, the car rental manager communicates the information about the availability of the car rental to the air ticket manager. In this case, the car rental transaction is the subject and the air ticket transaction is the observer. Now, based on the result the air ticket manager gets from the car rental manager, the former starts executing its subtransactions or does whatever is necessary to maintain the application semantics.

4.3.1 An Example Scenario

In this section we present an example scenario to illustrate the concepts behind execution dependencies, the communications that occur between the subtransaction managers, and transaction execution alternatives. A discussion on how the application and transaction semantics are translated into execution dependencies is also presented. Further, how these dependencies are used in producing multiple transaction execution alternatives is discussed.

The example includes a customer interested in planning a trip from Calgary to New York to attend a business meeting. However, at the same time there is a World Congress on Women's Issues being held in New York. His trip would include booking air tickets, renting a car and reserving an accommodation in New York. He specifies his preferences as far as these bookings and reservations are concerned. The entire trip transaction occurs based on the customer's preferences. At the same time he would also appreciate the best

alternatives he has to make this trip despite the expected overbooking of air tickets, accommodation and/or car rentals due to the World Congress.

Suppose the customer prefers to fly first class. However, under the given circumstances, an economy ticket would also serve the purpose. Further, assume he collects frequent flyer miles in Air Canada and hence prefers that carrier. He prefers a hotel or motel accommodation while he is in New York, but a hostel is unacceptable. As far as the car rental is concerned he only wants a sedan. Since he is an Avis-Advantage member he prefers Avis for car rental.

Further he specifies that he would like to book the ticket only if there is a car rental available. He also specifies that booking an air ticket does *not necessarily* depend on the type of accommodation available. He specifies that the air ticket may be booked if there is either a hotel or motel accommodation available. Further, he does not want to rent the car until he gets an accommodation reserved as per his specification. In essence, the semantics of this specification is that he wants to rent a car only if he has an accommodation confirmed. Further, he wants to book the air ticket either if he gets a car rental as per his specifications, or if he has an accommodation (hotel/motel).

In short, this describes the transaction's semantics and is used as an example of the expressive power of this model.

The above specifications suggest the following dependencies among the subtransactions that book the air tickets, reserve accommodation, and rent a car.

1. The car rental subtransaction depends on the successful completion of the accommodation subtransaction. In other words, only if the accommodation subtransaction yields a positive outcome does the car rental subtransaction start executing.
2. The subtransaction booking the air ticket is dependent on the successful completion of the car rental subtransaction. At the same time, it is not so strongly dependent on

the completion of the accommodation subtransaction. In other words, the air ticket subtransaction starts executing if either the car rental or the accommodation subtransaction succeeds.

The strengths of the dependencies identified above differ based on the specification.

The above specifications are translated into execution dependencies at an operational level when the transaction starts executing. The execution dependency databases located at each subtransaction manager level are populated with these dependencies. These dependencies differ in strengths and hence they populate the dependency database accordingly. This ensures the various transaction processing alternatives. At this point, the subtransaction managers identify every other subtransaction manager as a subject and/or an observer. For instance, the air ticket manager is an observer of the car rental and the accommodation subtransaction managers. Thus, the car rental and accommodation subtransaction managers are the subjects of air ticket subtransaction manager. Similarly, the accommodation subtransaction manager is a subject of the car rental subtransaction manager. In other words, the car rental subtransaction manager is an observer of the accommodation subtransaction manager.

The example illustrated here has dependencies between the subtransactions that book air tickets, reserve accommodation, and rent a car. The dependency database at the accommodation and car rental subtransaction managers is shown in Figures 4.2 and 4.3, respectively. These dependencies are deduced from the specifications set for the trip transaction execution. The transaction is divided into three subtransactions – to book an air ticket, reserve an accommodation and rent a car. Each of these subtransactions may be further divided to achieve the desired result. For instance, the accommodation transaction is further divided to obtain a reservation in a hotel, motel or hostel. Similarly the air ticket subtransaction could be further divided to book either an economy or a first class ticket. In this example, all the dependency databases are present in nesting level L2 (see Figure 4.1, Page 67). Hence all the subtransactions at that level are referred to as subtransaction managers. For instance, the air ticket subtransaction is referred to as an air ticket

manager. These managers play the role of subjects and/or observers depending on the transaction execution. The transaction execution is controlled by the dependency information stored in the dependency databases.

Subject	*Observer*	*Dependency*
Accommodation Manager	*Car Rental Manager*	*Necessary*
Accommodation Manager	*Air Ticket Manager*	*Sufficient*

Figure 4.2 Dependency Database at Accommodation Manager

Subject	*Observer*	*Dependency*
Car Rental Manager	*Air Ticket Manager*	*Necessary*

Figure 4.3 Dependency Database at Car Rental Manager

In our example, it is evident from the specifications that the accommodation subtransaction must successfully complete so that the other subtransactions can execute. Though all the subtransactions are simultaneously submitted, the rental and the ticket subtransactions wait until the accommodation subtransaction completes successfully and receive notification of the same. Hence the accommodation subtransaction is a subject from the perspectives of air ticket and car rental subtransactions. In other words, the air ticket and car rental subtransactions are observers of the accommodation subtransaction. It is evident from the dependency database at the car rental manager that the air ticket transaction is *necessarily* (N) dependent on the successful completion of the rental transaction. Similarly, the dependency database at the accommodation manager shows that the rental transaction is *necessarily* (N) dependent, while the air ticket transaction is only *sufficiently* (F) dependent, on the accommodation subtransaction. This implies that the air ticket manager starts executing the air ticket subtransaction as soon as it receives the notification from the accommodation manager. It does not wait for the car rental manager's notification though the air ticket subtransaction has a *necessary-type* dependency with the car rental subtransaction. This is because in our transaction model

the sufficient-type dependency takes priority over any other type of dependency (see discussion on Execution Dependencies in Chapter 3).

As mentioned above the various subtransactions execute with respect to the dependency they have with other subtransactions. In addition, all the other transaction execution alternatives are also pursued to provide the user with a wide range of results. For instance, though the customer prefers hotel or motel, our transaction model runs the hotel, motel, and hostel subtransactions, and provides him with all the results available. This is useful when a particular transaction execution fails but some other execution provides an *equivalent* result. Similarly, suppose there are no seats available in the first class due to the World Congress on Women's Issues. However if there are seats in the economy class, then those results are presented to the customer. Now he could use these results to book his tour. Though the customer prefers only a first class ticket, he might be willing to go on the trip on an economy ticket based on his "necessity" rather than his "preference", under the given circumstances.

4.4 A Comparison With Conventional Transaction Models

The advanced transaction models proposed in the past have been suitable for certain application domains. However, each model has its shortcomings. This thesis discusses the shortcomings of the earlier models only with respect to the concurrency aspect of transaction management. Further, it takes into account the ACID properties of the transactions executing within the frameworks of those models. The suitability of these models in an MDBS is also discussed wherever applicable.

A closed nested transaction model rules out the possibility of relaxing ACID properties because of the subtransactions. The subtransactions execute in a "closed" fashion and hence do not reveal partial results. This results in less concurrency among subtransactions. An open nested transaction model enhances the parallelism among the subtransactions. In this model, all the subtransactions reveal their results to other subtransactions. This results in violation of the isolation property. In other words, the ACID properties are relaxed. However, past research indicates that when this model is

applied as such to an MDB environment, it does not help in maintaining the autonomy of the underlying system. The transaction model presented in this thesis is a variation of an open nested model with additional features. Our model extracts the application semantics and accordingly relaxes the ACID property. At the same time, it allows the characterization of the autonomy interface of the underlying MDB environment by appropriately utilizing the application semantics in the form of execution dependencies and subtransaction managers.

Sagas address the delays in transaction processing due to the long-lived nature of the transactions. A long-lived transaction can be expressed as a saga if it can be written as a sequence of transactions that can be interleaved with other transactions. Partial executions are undesirable and they must be undone.

When sagas are applied to MDBSs, local autonomy is not severely violated because each element database sees each subtransaction as a local transaction managed by the element DBMS. This finds application only in environments where long-lived, compensatable transactions execute. Providing compensating transactions is a major difficulty in the case of sagas. They cannot be applied in scenarios where a transaction is irreversible, such as drilling holes. Nevertheless, sagas are appealing to compensatable MDBS environments because they affect the autonomy of the underlying systems minimally.

A saga only permits two levels of nesting unlike the nested transaction presented in this thesis. Sagas also compromise on the atomicity at the top level thereby allowing other sagas to view their partial results unlike the nested transaction model presented here which maintains the atomicity at the global level. Further, sagas do not provide multiple transaction execution alternatives as the model presented here does.

The Flex transaction model is the closest match to the model presented in this thesis. It identifies and addresses the challenge of maintaining the autonomy of the underlying systems in a MDB environment. Specifically designed for MDB environments, this transaction model shares our goal of providing multiple transaction execution alternatives

based on execution dependencies. It decomposes a global transaction into several functionally equivalent subtransactions as our model does. However, our model identifies and presents a broader range of execution dependencies than the Flex transaction model, which identifies only two extreme types of dependencies namely, positive and negative dependencies.

Flex transaction model has provisions for the execution of both compensatable and non-compensatable transactions. This thesis assumes reliability and hence does not explicitly contribute provisions for compensatable transactions. However, by taking into account the dynamics of both the environment and transaction processing, we can easily extend this work to address compensatable transactions.

Apart from the specific shortcomings mentioned above, none of the models appear to be scalable to run applications in the MDB environments in the Internet. The implementation of our paradigm is readily scalable to MDB environments in the Internet.

In summary, the shortcomings of the various models are:

- Absence of a mechanism to specify and provide proper global integrity constraints (dependencies) that determines the effects on the global atomicity,
- Unsuitability of certain transaction models in MDB environments,
- A means to characterize the amount of local autonomy affected,
- A mechanism to exploit the application semantics and thereby provide multiple transaction execution alternatives, and
- Scalability to Internet environments that run advanced database applications.

A brief discussion of the concepts that address the above issues is deferred to Chapter 5.

4.5 Summary

This chapter presented the analysis of the transaction management problem in multidatabases. A discussion of our transaction model was presented and the concepts

underlying the model were revisited in Section 4.2. Section 4.3 introduced an application to illustrate the ideas underlying our transaction model. Finally, we presented a comparison of the transaction paradigm presented here with the conventional models. Chapter 5 presents the conclusion and summary of the various concepts discussed in this dissertation and sets some future research directions.

In my end is my beginning.

- *T. S. Eliot, Four Quartets.*

Chapter 5
Conclusions and Future Work

This dissertation concludes with a summary of its contributions and directions for future work.

5.1 Summary of Contributions

This thesis identifies the following problems that affect the management of nested transactions in multidatabase systems. An implementation of a nested transaction model that describes the pragmatic components required to realize the following features in the form of an abstract model was presented.

- Absence of a mechanism to identify and exploit dependencies among subtransactions.

  Conventional transaction models lack a mechanism by which they can identify and exploit dependencies in an application running in a MDB environment. Today, in most applications, the execution of transactions is directed by application semantics. A mechanism to identify the application semantics and exploit the same can enhance

the parallelism of transaction processing. The lack of such a mechanism affects the concurrency aspects of transaction management.

- Lack of a mechanism to maintain the autonomy of the underlying systems.

  Though there have been numerous proposals to address transaction management in distributed systems, many of those models have found little use when applied to multidatabase systems. This is due to the severe effects they have on the autonomy of the underlying systems in a MDB environment. The maintenance of autonomy to the best maximum level is imperative when addressing the transaction management problem in a MDBS.

- Absence of a mechanism to communicate partial results among subtransactions thereby increasing parallelism.

  Most advanced transaction models are generalizations of the nested transaction model. The transactions in these generalizations usually attempt to relax the ACID paradigm by allowing partial results to be exposed to their subtransactions. However, a better mechanism to implement such visibility rules is important to enhance the parallelism of transaction execution in a MDBS. The crux is when it becomes important that the autonomy of the underlying systems must also be maintained in addition to the maintenance of a proper visibility mechanism.

- Lack of a mechanism to produce multiple transaction execution alternatives.

  The advanced transaction models proposed in the past produce transaction execution that strictly adheres to the requirement of the end user. However, in most current day applications, it is important that there be multiple transaction execution alternatives so that even if the results of one execution is unsatisfactory, the user can choose from other alternatives presented by the system.

Our research analyzed the above problems and/or challenges and addressed them using a novel implementation of an open nested transaction in a multidatabase environment characterized with complete autonomy. The implementation provides a framework that can be utilized by several advanced database applications.

- **Execution Dependencies:** The application and transaction semantics are translated into dependencies existing among subtransactions. Such dependencies are called execution dependencies because they direct the execution of transactions. This thesis identifies three different types of execution dependencies based on the application and transaction semantics in advanced database applications. They are *necessary*, *sufficient* and *bonus* execution dependencies. These dependencies are the global integrity constraints of the transaction processing system. The definitions and details of the various dependencies are discussed in Chapter 3.

- **Autonomy Interface:** The underlying system considered in this thesis is a multidatabase system. It is characteristic of a multidatabase environment to be autonomous. Hence, it becomes important to maintain the level of autonomy maximally. This thesis addresses the issue through the concept of execution dependency database at various subtransaction managers. The execution dependency database at a subtransaction consists of the various execution dependencies it has with other subtransactions. It is based on the dependencies in the execution dependency database that the transaction executes in certain specific ways. This acts as an autonomy interface because, the higher the level of this interface in a nesting, the more autonomous is the underlying system, and vice versa. The execution dependency databases at the subtransaction managers are populated dynamically and hence, at an operational level, the transaction model has a layered architecture.

- **Subjects and Observers:** ACIDity in a transaction is relaxed by allowing the exposure of partial results to relevant subtransactions. This has been successfully

shown in many open nested transaction models. However, this affects the autonomy of the underlying system if the system under consideration is a MDBS. This thesis addresses the issue by borrowing the concepts of behavioral design patterns. Specifically, it identifies the subtransaction managers and based on the execution dependency available at these managers, it dynamically categorizes the subtransaction managers as subjects and observers. Any subtransaction manager can play both these roles as long as the transaction processing adheres to the application and transaction semantics. Hence, based on the dependency information at a subtransaction manager's execution dependency database, it communicates the results to all other subtransaction managers that have been dynamically included as its observers. This is an effective way of communicating information among the subtransactions whereby multiple observers of a subtransaction get to know the results of a subtransaction being observed.

- **Multiple Transactions, Same Global Objective:** The utilization of the execution dependencies and the concepts of subjects and observers yield a transaction execution that satisfies the application semantics specified. However, the paradigm developed in this thesis produces multiple outcomes for the same global objective using functionally equivalent subtransactions thereby widening the range of choice of the outcomes. This helps in situations where even if one execution fails, the user can rely on the various alternatives the system produces.

5.2 Future Directions

There are several interesting directions in which the work presented in this dissertation can proceed. The future work suggested here is based on this work coupled with directions to address the general problem of transaction management with respect to Internet and other wireless technologies.

Reliability and Heterogeneity: In this dissertation we have investigated the various aspects of transaction management in multidatabase systems. However, we assumed transaction management from the perspective of completely reliable systems. Though the

assumption is valid from the perspective of academic research, it is not the case with systems in the real world. Similarly, we have considered a homogeneous system to demonstrate the paradigm presented in this dissertation. Recommendations for future work include the enhancement of the model presented in this dissertation by considering a heterogeneous environment with reliability problems.

Scalability to Distributed Internet Applications: The transaction paradigm presented in this dissertation addresses the transaction management problem in multidatabase systems. Though its implementation is readily scalable to Internet environments, we believe that such scalability could come with additional problems. This could be due to the truly distributed nature of the Internet environment characterized with total autonomy and a high level of heterogeneity. Such environments pose interesting problems that can be addressed using the fundamental principles of transaction management in traditional environments coupled with the new principles of transaction management in truly electronic environments. This is a highly potential area because the distributed environments today are fully Internet-enabled or are in the process of being enabled.

Application in Wireless Environments: An interesting direction for future work would be in the area of mobile databases and wireless transaction management. Most applications today are developed with the consideration for potential use in wireless environments or on wireless devices. Though these applications are in their early stage, the day is not far when transactional aspects would be incorporated into such devices. A study about the feasibility of our paradigm in such environments could be a direction of future work. This would bring in a whole new world of opportunities in the *'at anytime, from anywhere'* concept of wireless transaction research. An example of this area is the new paradigm in the online business world of *m-commerce (mobile-commerce)*. This area of research would pose a lot of technical challenges made interesting due to the wireless nature of the environments. Hence, we recommend this area be investigated as part of the future work.

If I have seen farther, it is by standing on the shoulders of Giants.

- Sir Isaac Newton, letter to Sir Robert Hooke, Feb. 5, 1676.

Bibliography

[AVA+94] G. Alonso, R. Vingralek, D. Agrawal, Y. Breitbart, A. El Abbadi, H-J. Schek, G. Weikum, "Unifying Concurrency Control and Recovery of Transactions", *Information Systems Vol. 19, No. 1, pp. 101-115, 1994.*

[B90] K. Barker, "Transaction Management on Multidatabase Systems", *Ph.D. Thesis, Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada, 1990.*

[B94] K. Barker, "Quantification of Autonomy on Multidatabase Systems", *Journal of Systems Integration, 4, 1994, Pg.: 151 – 169.*

[BHG87] P.A. Bernstein, V. Hadzilacos, N. Goodman, "Concurrency Control and Recovery in Database Systems", *Addison-Wesley, 1987.*

[BÖ90] K. Barker, M.T. Öszu, "Concurrent Transaction Execution in Multidatabase Systems", *Proc. Of COMPSAC' 90. The 14th Annual International Computer Software and Applications Conference, 1990, Pg.: 282 – 288.*

[BE99] K. Barker, A. Elmagarmid, "Transaction Management in Multidatabase Systems: Current Technologies and Formalisms", in Collection, *"Management of Heterogeneous and Autonomous Database Systems", Morgan Kaufmann Publishers, 1999. (Edited by: A. Elmagarmid, M. Rusinkiewicz, and A. Sheth)*

[BBE99] A. Bouguettaya, B. Benetallah, A. Elmagarmid, "An Overview of Multidatabase Systems: Past and Present", in Collection, *"Management of Heterogeneous and Autonomous Database Systems", Morgan Kaufmann Publishers, 1999. (Edited by: A. Elmagarmid, M. Rusinkiewicz, and A. Sheth)*

[BST90] Y. Breitbart, A. Silberschatz, G.R. Thompson, "Reliable Transaction Management in a Multidatabase System", *Proc. Of ACM-SIGMOD International Conference On Management of Data, 1990, Pg.: 215 – 294.*

[CR90] P. Chrysanthis, K. Ramamritham, "ACTA: A Framework for Specifying and Reasoning about Transaction Structure and Behavior", *Proc. Of the ACM SIGMOD International Conference On Management of Data, 1990, Pg.: 194 – 203.*

[CR91] P. Chrysanthis, K. Ramamritham, "A Formalism for Extended Transaction Models", *Proc. Of the 17th International Conference on Very Large Databases, 1991, Pg.: 103 – 112.*

[CR94] P. Chrysanthis, K. Ramamritham, "Synthesis of Extended Transaction Models Using ACTA", *ACM Transactions on Database Systems, Vol. 19, No.3, 1994, Pg.: 450 – 491.*

[CR99] P. Chrysanthis, K. Ramamritham, "Correctness Criteria and Concurrency Control", in Collection, *"Management of Heterogeneous and Autonomous Database Systems", Morgan Kaufmann Publishers, 1999. (Edited by: A. Elmagarmid, M. Rusinkiewicz, and A. Sheth)*

[DE89] W. Du, A. Elmagarmid, "Quasi-serializability - a Correctness Criterion for Global Concurrency in InterBase", *Proc. Of the 15th International Conference on Very Large Databases, Amsterdam, 1989, Pg.: 347 – 355.*

[E92] A. Elmagarmid (Ed.) "Database Transaction Models for Advanced Applications", *Morgan Kaufmann Publishers, 1992.*

[EB97] S.A. Ehikioya, K. Barker, "A Formal Specification Strategy for Electronic Commerce", *Proc. Of IDEAS' 97. The International Database Engineering and Applications Symposium, Montreal, 1997, Pg.: 201 – 210.*

[ELL+90] A. Elmagarmid, Y. Leu, W. Litwin, M. Rusinkiewicz, "A Multidatabase Transaction Model for InterBase", *Proc. Of the 16th International Conference on Very Large Data Bases, 1990, Pg.: 507 – 518.*

[ERS99] A. Elmagarmid, M. Rusinkiewicz, A. Sheth (Ed.) "Management of Heterogeneous and Autonomous Database Systems", *Morgan Kaufmann Publishers, 1999.*

[EW99] S.A. Ehikioya, T. Walowetz, "A Formal Specification of Transaction Systems in Distributed Multi-Agents Systems", *Proc. Of the ISCA. 14th International Conference, Cancun, 1999, Pg.: 378 – 383.*

[FÖ89] A.A. Farrag, M.T. Özsu, "Using Semantic Knowledge of Transactions to Increase Concurrency", *ACM Transactions on Database Systems, Vol. 14, No.4, 1989, Pg.: 503 – 525.*

[G81] J.N. Gray, "The Transaction Concept: Virtues and Limitations", *Proc. Of the 7th International Conference on Very Large Data Bases, 1981, Pg.: 144 – 154.*

[G83] H. Garcia-Molina, "Using Semantic Knowledge for Transaction Processing in a Distributed Database", *ACM Transactions on Database Systems, Vol. 8, No.2, 1983, Pg.: 186 – 213.*

[GGK+91] H. Garcia-Molina, D. Gawlik, J. Klein, K. Kleissner, K. Salem, "Modeling Long-Running Activities as Nested Sagas", *IEEE Data Engineering Bulletin, Vol. 14, 1991, Pg.: 14 –18.*

[GR93] J.N. Gray, A. Reuter, Transaction Processing: Concepts and Techniques. *Morgan Kaufmann Publishers, 1993.*

[GS87] H. Garcia-Molina, K. Salem, "Sagas", *Proc. Of the ACM-SIGMOD Annual Conference, 1987, Pg.: 249 – 259.*

[GHJ+95] E. Gamma, R. Helm, P. Johnson, J. Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software", *Addison-Wesley, 1995.*

[H88] V. Hadzilacos, "A Theory of Reliability in Database Systems", *Journal of the ACM, Vol. 35, No. 1, 1988, Pg.: 121 – 145.*

[HR87] T. Haerder, K. Rothermel, "Concepts of Transaction Recovery in Nested Transactions", *Proc. Of ACM-SIGMOD, CA, 1987, Pg.: 234 – 248.*

[KR93] M. Kamath, K. Ramamritham, "Performance Characteristics of Epsilon Serializability with Hierarchical Inconsistency Bounds", *Proc. Of the 9th International Conference Of Data Engineering, Austria, IEEE Computer Society, 1993, Pg.: 587 – 594.*

[L83] N. Lynch, "Multi-level Atomicity – A New Correctness Criterion for Database Concurrency Control", *ACM Transactions on Database Systems, Vol. 8, No.2, 1983, Pg.: 484 – 502.*

[L92] D.B. Lomet, "MLR: A Recovery Method for Multi-level Systems", *Proc. Of ACM-SIGMOD International Conference On Management of Data, 1992, Pg.: 185 – 194.*

[LHL97] S. Lee, C. Hwang, W. Lee, "A Uniform Approach to Global Concurrency Control and Recovery in Multidatabase Environment", *Proc. Of the Sixth International Conference on Information and Knowledge Management, 1997, Pg.: 51 – 58.*

[M81] J.E.B. Moss, "Nested Transactions: An Approach to Reliable Distributed Computing", *M.I.T Report, MIT/LCS/TR-260, MIT, Laboratory of Computer Science, 1981.*

[M85] J.E.B. Moss, "Nested Transactions: An Approach to Reliable Distributed Computing", *MIT Press, 1985.*

[MRK+91] S. Mehrotra, R. Rastogi, H.F. Korth, A. Silberschatz, "Non-Serializable Executions in Heterogeneous Distributed Database Systems", *Proc. Of the First International Conference on Parallel and Distributed Systems, 1991.*

[MRB+92] S. Mehrotra, R. Rastogi, Y. Breitbart, H.F. Korth, A. Silberschatz, "The Concurrency Control Problem in Multidatabases: Characteristics and Solutions", *Proc. Of ACM-SIGMOD International Conference on Management of Data, 1992, Pg.: 288 – 297.*

[ÖV99] M.T. Özsu, P. Valduriez, "Principles of Distributed Database Systems, 2/e", *Prentice Hall, 1999.*

[P91] C. Pu, "Generalized Transaction Processing with Epsilon Serializability", *Proc. Of 4th International Workshop on High Performance Transaction Systems, 1991.*

[PL90] C. Pu, A. Leff, "Epsilon Serializability", *Technical Report CUCS-054-90, Dept. of Computer Science, Columbia University, 1996.*

[PL91] C. Pu, A. Leff, "Replica Control in Distributed Database Systems: An Asynchronous Approach", *Proc. Of ACM-SIGMOD International Conference On Management of Data, CO, 1991, Pg.: 377 – 386.*

[PL92] C. Pu, A. Leff, "Autonomous Transaction Execution with Epsilon Serializability", *Proc. Of RIDE Workshop on Transaction and Query Processing, AZ, IEEE Computer Society, Pg.: 2 –11.*

[R78] R. Reed, "Naming and Synchronization in a Decentralized Computer System", *M.I.T Report, MIT/LCS/TR-205, MIT, Laboratory of Computer Science, 1978.*

[R89] A. Reuter, "Contract: A Means For Extending Control Beyond Transaction Boundaries", *Proc. Of the Third International Workshop on High Performance Transaction Systems, 1989.*

[RC96] K. Ramamritham, P.K. Chrysanthis, "A Taxonomy of Correctness Criteria in Database Applications", *VLDB Journal, Vol. 5, No. 1, 1996, Pg.: 85 – 97.*

[RKT+95] M. Rusinkiewicz, W. Klas, T. Tesch, J. Wasch, P. Muth, "Towards a Cooperative Transaction Model – The Cooperative Activity Model", *Proc. Of the 21st Conference of Very Large Data Bases, 1995, Pg.: 194 – 205.*

[RP95] K. Ramamritham, C. Pu, "A Formal Characterization of Epsilon Serializability", *IEEE Transactions on Knowledge and Data Engineering, Vol. 7, No. 6, 1995, Pg.: 997 – 1007.*

[SAS99] H. Schuldt, G. Alonso, H-J. Schek, "Concurrency Control and Recovery in Transactional Process Management", *Proc. Of the ACM Symposium on Principles of Database Systems (PODS'99), 1999, Pg.: 316 – 326.*

[SL90] A. Sheth, J. Larson, "Federated Database Systems for Managing Distributed Heterogeneous, and Autonomous Databases", *ACM Computing Surveys, Vol. 22, No. 3, 1990, Pg.: 183 – 236.*

[SWY93] H.-J. Schek, G. Weikum, H. Ye, "Towards a Unified Theory of Concurrency Control and Recovery", *Proc. Of the ACM Symposium on Principles of Database Systems (PODS'93), 1993Pg.: 300 – 311.*

[VHB+98] R. Vingralek, H. Hasse, Y. Breitbart and H.-J. Schek, "Unifying Concurrency Control And Recovery of Transactions With Semantically Rich Operations", *Journal of Theoretical Computer Science 190 (1998) pg. 363-396, 1998.*

[W86] G. Weikum, "A Theoretical Foundation of Multi-level Concurrency Control", *Proc. Of 5th ACM-SIGACT-SIGMOD Symposium on Principles of Database Systems, 1986, Pg.: 31 – 42.*

[WHB+90] G. Weikum, C. Hasse, P. Broessier, P. Muth, "Multi-level Recovery", *Proc. Of 9th ACM-SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, 1990, Pg.: 109 – 123.*

[W91] G. Weikum, "Principles and Realization Strategies of Multilevel Transaction Management", *ACM Transactions on Database Systems, Vol. 16, No. 1, 1991, Pg.: 132 – 180.*

[WYP97] K. Wu, P.S. Yu, C. Pu, "Divergence Control Algorithms for Epsilon Serializability", *IEEE Transactions on Knowledge and Data Engineering, Vol. 9, No. 2, 1997, Pg.: 262 – 274.*

[WS92] G. Weikum, H. -J. Schek, "Concepts and Applications of Multilevel Transactions and Open Nested Transactions", in Collection, *"Database Transaction Models for Advanced Applications", Morgan Kaufmann Publishers, 1992. (Edited by: A. Elmagarmid)*