AN EXTENSIBLE APPROACH TO REALIZING EXTENDED TRANSACTION MODELS

By

EMAN ANWAR

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

1996

ACKNOWLEDGEMENTS

I am deeply indebted to my advisor Dr. Sharma Chakravarthy for providing me the opportunity to work on this challenging topic and for providing continuous guidance, advice, and support throughout the course of this research work. I thank Dr. Stanley Su, Dr. Eric Hanson, Dr. Li-Min Fu and Dr. Suleyman Tufekci for serving on my supervisory committee and for their careful perusal of this dissertation. I would like to thank Sharon Grant for maintaining a well administered research environment and for the great laughs we had together. I also thank many of my friends for making my stay in Gainesville memorable.

On a more personal note, I would like to dedicate this dissertation to my father, Dr. Sami Anwar and my mother, Mrs. Afaf Anwar, who have always encouraged my academic interests. I would also like to thank my two wonderful sisters, Hanan and Nahla, and my cute little nephew, Tamer. Without their love, support, and constant encouragement this work would not have been possible.

TABLE OF CONTENTS

LIST OF FIGURES

Abstract of Dissertation Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Doctor of Philosophy

AN EXTENSIBLE APPROACH TO REALIZING EXTENDED TRANSACTION MODELS

By

Eman Anwar

May, 1996

Chairman: Dr. Sharma Chakravarthy
Major Department: Computer and Information Science and Engineering

Use of databases for non-traditional applications has prompted the development of an array of new transaction models whose semantics vary from the traditional model as well as from each other. The implementation details of most of the proposed models have been sketchy at best. Furthermore, current architectures of most DBMSs do not lend themselves to supporting more than one *built-in* transaction model. As a result, despite the presence of rich transaction models, applications cannot realize semantics other than that provided by the traditional transaction model.

In this dissertation, we propose a framework for supporting various transaction models in an *extensible* manner. We demonstrate how ECA (event-condition-action) rules, defined at the *system level* on significant operations of a transaction and/or data structures such as a lock table, allow the database implementor/customizer to support: i) currently proposed extended transaction models, and ii) newer transaction models as they become available. Most importantly, this framework allows one to customize transaction (or application) semantics in arbitrary ways using the same underlying mechanism. *Sentinel*, an active object-oriented database system developed at UF, is used for demonstrating our approach for implementing extended transaction models.

CHAPTER 1
INTRODUCTION

Database systems allow for the storage, management and manipulation of large volumes of data. Users interact with the database system by executing *applications*, where each application consists of a set of operations to be performed on the stored data. These sets of operations carry out a desired function on behalf of the user, e.g., debiting or crediting a bank account. An application's execution yields a partially ordered set of *read* and *write* operations. In other words, an application interacts with the database only through the read and write operations. These latter operations respectively denote the reading and writing of a data item's value.

As stated above, an application's execution produces a partially ordered set of read and write operations. This partially ordered set is referred to as a *transaction.* Conventional database management systems (DBMSs) guarantee four properties for each transaction, namely, *atomicity, consistency, isolation* and *durability* (commonly referred to as the ACID properties) [25, 26, 39]. These four properties are associated with transactions in order to preserve the correctness of the database as well as allow certain database system features to be provided. For instance, in order to allow users to concurrently access the database, a mechanism for regulating and governing the execution of their transactions needs to be employed. This mechanism is required since these concurrent transactions access a *shared* database. The isolation property was associated with transactions in order to achieve that precise goal, namely, the correct execution of concurrent transactions. These four properties evolved the first and simplest transaction model, namely, the *traditional transaction model.* A transaction model is basically a set of rules which govern the execution and properties of transactions. Below, we give a detailed discussion of the need for associating these properties with transactions. In addition, examples of bad scenarios which arise when these properties are not enforced are given.

1.1 Need for Atomicity

A transaction is a linear sequence of operations against the database. During the course of a transaction's execution, one of these operations may fail or a system failure may occur thereby preventing the transaction from continuing its execution. The problem with failures can be demonstrated by considering a transaction which transfers \$200 from a person's savings account to the person's checking account. A transaction which performs this transfer of funds can be modeled as follows:

```
Line 1   temp = read(savings);  /* read savings balance into variable */
Line 2   temp = temp - 200;
Line 3   write(temp,savings);   /* decrement the savings bank balance */
Line 4   temp = read(checking); /* read checking balance into variable */
Line 5   temp = temp + 200;
Line 6   write(temp,checking);  /* increment the checking bank balance */
```

Assume a system failure occurs after execution of line 3 above, specifically, after decrementing the savings account but before incrementing the checking account. In this scenario, the database does not reflect a correct transfer of funds; rather, the funds are lost and not placed into the checking account. This is caused because only a subset of the operations (instead of all) of the transaction is executed. Therefore, it is necessary to ensure that all of the operations of a transaction are executed. Furthermore, in the event of failures it must appear that none of the operations have taken place. In other words, a transaction must appear to be one atomic logical unit; either all or none of the operations are executed. Thus, a transaction must take the database from one initial database state to a result state without any observable intermediate states. Furthermore, in the event of errors, the transaction must appear to have not changed the initial database state. A transaction which performs all or none of its operations is termed *atomic*.

1.2 Need for Consistency

A database consists of a collection of data items which satisfy a set of integrity constraints. For example, a possible integrity constraint on a bank balance is that it's value should always be greater than or equal to zero. Another integrity constraint may specify that all employees should have a salary greater than or equal to the minimum wage. The notion of integrity constraints raises an important issue, in particular, how to ensure the preservation of integrity constraints despite the continual modification of data by transactions.

Two schools of thought exist for the above mentioned problem. The first method places the burden on the application programmer; specifically, the application programmer must be aware of all integrity constraints that exist in the database and write application programs which preserve them. On the other hand, the second method removes the burden from the application programmer and places it with the database system. In particular, after each transaction completes, the database system is responsible for checking that no integrity constraints are violated. This is by no means an easy task since the database needs to be quiescent while the integrity constraints are checked. This in turn reduces the availability of the database to the users. Moreover, a substantial overhead is incurred; the database system must check that all integrity constraints are not violated after each transaction completes. The disadvantages of the latter approach outweigh the advantage of removing the burden of integrity constraint preservation from the user. Consequently, the traditional transaction model dictates that each transaction, written by an application programmer, is correct. A transaction is deemed correct if it takes the database from a state, in which the integrity constraints are satisfied, to another state where the integrity constraints are also preserved. This correctness property of transactions is referred to as *consistency*. It is important to note that although consistency is preserved once a transaction successfully terminates, consistency may be violated during the intermediate steps of a transaction.

1.3 Need for Isolation

Another property which a transaction must possess is the *isolation* property. Transactions consist of a collection of *read* and *write* operations which respectively retrieve and update the data. One of the primary goals of a database system is to allow for the simultaneous execution of several transactions against a database, either by the same user or by multiple different users. However, the execution of concurrent transactions needs to be controlled and coordinated since they are accessing a shared database and may potentially interfere with one another. In other words, these concurrent transactions must be executed while preserving the illusion that each transaction is executing in isolation. In order to show the need for this property, consider two identical banking applications which withdraw the sum of \$100 from the same bank account. Furthermore, assume that the initial balance is \$500 and that each transaction must first retrieve the bank balance before updating it. Therefore, each transaction is structured as follows:

```
Line 1   temp = read(balance);      /* read bank balance into variable */
Line 2   temp = temp - 100;
Line 3   write(temp,balance);       /* update the bank balance */
```

Now consider the following interleaved execution of the above transaction:

```
Transaction One                              Transaction Two

temp = read(balance);
                                              temp = read(balance);
```

```
                                            temp = temp - 100;

temp = temp - 100;

write(temp,balance);

                                            write(temp,balance);
```

In the above interleaved execution, both transactions read the initial bank balance of \$500 and decrement it by \$100 and then both write the bank balance as \$400, i.e., only a withdrawal of \$100 rather than \$200 is reflected in the database. This problem is referred to as the *lost update* or *missing update* problem.

Another problem which may potentially arise when concurrent transactions execute is known as the *dirty read* problem. This problem arises when a transaction T1 reads the value of a data item X which was previously written by a transaction T2. Subsequently, T2 makes additional modifications to that same object X. Consequently, transaction T1 has read an intermediate rather than a final value of object X. To given an example of when the dirty read problem arises, consider the following scenario. Assume an employer is to be given a 10% raise and a bonus in the amount of \$300. A transaction performing this salary increase can be structured as follows :

```
temp = read(salary);       /* read salary into temporary variable */

temp = temp + temp * 0.1;

write(temp,salary);        /* update the salary */

temp = temp + 300;

write(temp,salary);        /* update the salary with the bonus*/
```

Assume another transaction reads the employer's salary after execution of line 3. In this case the incorrect *final* salary will be read thus introducing a dirty read.

The last problem which may arise when transactions execute concurrently is termed the *unrepeatable read* problem. This problem arises when a transaction T1 reads an object X once before transaction T2 modifies it, and once after transaction T2 modifies it and commits. In this scenario, the reads made by transaction T1 produce different results. Consequently, the first read made by transaction T1 is termed *unrepeatable* since its value cannot be repeated since a new value of object X is returned. The problems associated with concurrent execution of transactions are depicted in Figure 1.1. <O,V> denotes the reading of object O and the return of value V or updating object O's value to V.

Lost Update			**Dirty Read**			**Unrepeatable Read**		
T2	READ	<O,1>	T2	WRITE	<O,2>	T2	READ	<O,1>
T1	WRITE	<O,2>	T1	READ	<O,2>	T1	WRITE	<O,2>
T2	WRITE	<O,3>	T2	WRITE	<O,3>	T2	READ	<O,2>

Versions of an object O are shown to be read and written, in an interleaved fashion, by two tranasactions, T1 and T2. The WRITE-WRITE dependency can cause lost updates, the WRITE-READ dependency can cause dirty reads, and the READ-WRITE dependency can cause unrepeatable reads.

Figure 1.1. Transaction Dependencies caused by Concurrent Transaction Execution.

From the above discussion, it is necessary to either execute only one transaction at a time or allow multiple transactions to execute concurrently but create the illusion that each transaction executes in isolation from other transactions. The former approach has two main disadvantages. First, the performance of the database system, in terms of transaction throughput, is severely compromised since only one transaction is allowed to execute at a time. Second, many transactions may be independent from each other, specifically, the transactions may access disjoint sets of data items. When transactions are independent, their concurrent executions do not interfere with each other. Due to these two main disadvantages, it is more advantageous to allow the concurrent execution of transactions, but regulate it in such a manner such that each transaction appears to execute in isolation from other transactions. The property of creating the illusion that concurrent transactions execute in isolation from one another is termed *isolation.*

1.4 Need for Durability

The last property of traditional transactions is *durability.* This property ensures that all updates made by a committed transaction survive any subsequent failures. This property is necessary since it is impossible to build a perfect system that never fails. Even if we are to assume that such a system can be built, the system will still fail occasionally. Failures will occur due to users making application programming errors as well as the people operating the system making some procedural or data entry errors causing the system to fail. Consequently, since an error free system is infeasible, durability is required to ensure that when a transaction commits, all updates survive any subsequent failures.

Below, we summarize the ACID properties of transactions in the traditional transaction model:

- **Atomicity.** This property refers to the fact that all or none of the operations of a transaction are executed. In other words, the transaction is an atomic unit of work.
- **Consistency.** This property ensures that a transaction preserves the consistency of the database, i.e., the transaction takes the database from one consistent state to another. Note, however, that the database may be inconsistent at intermediate points within a transaction.
- **Isolation.** This property ensures that concurrent transactions are isolated from one another. In other words, the partial results of a transaction T1 are hidden from all

other concurrent transactions that interfere with T1, until transaction T1 commits. Consequently, from the perspective of a transaction, its execution behaves exactly as it would in single-user mode as it does in multi-user mode.

- **Durability.** This property ensures that the updates made by a committed transaction survive subsequent failures. In other words, the results of a committed transaction can be reestablished despite any subsequent user, environment or hardware component failure. This property also implies that there is no automatic function for revoking the effects of a completed and committed transaction. These effects can only be revoked by executing a counter-algorithm which is referred to as a *compensating transaction*. Compensating transactions will be discussed in later sections.

Although database systems encompass a large range of applications, they were originally developed for business oriented database applications such as banking systems, airline reservation systems, and organizational systems. These applications are well-served by the properties of the traditional transaction model. However, as the scope of databases extends to a large variety of applications, it is important to reevaluate the assumptions and properties of the traditional model of transactions. These newer and non-traditional applications expose some limitations of the traditional transaction model. It must be emphasized that although the traditional transaction model has some limitations, it has many virtues as well. One of the primary advantages of the traditional transaction model is its simplicity which makes transactions with ACID properties effective in isolating the application from failures and faults. In the following section we examine the limitations of the traditional transaction model. Specific applications which make these limitations apparent are also discussed.

1.5 Limitations of the Traditional Transaction Model

Conventional database management systems (DBMSs) guarantee atomicity, consistency, isolation and durability (commonly referred to as the ACID properties) [25, 26, 39] for each transaction. This traditional transaction model is ideal for applications in areas such as airline reservations, electronic funds transfer and banking applications. Transactions for these applications are typically simple in nature and have very short duration. However, the emergence of non-traditional applications, such as workflow management, cooperative tasks, and computer integrated manufacturing (CIM), has made it increasingly apparent that this traditional transaction model is too restrictive for modeling these newer applications. Transactions in these newer applications differ considerably from transactions in areas such as banking applications; specifically, transactions are usually very complex, access many data items during the course of their execution and have long duration. Transactions having long duration are referred to as *long-lived* transactions.

The traditional transaction model is inadequate for serving the requirements of long-lived transactions. First, long-lived transactions are more prone to failures due to their long duration. The failure atomicity requirement of the traditional transaction model dictates that all work must be rolled back in the event of failures. This requirement is unsuitable for long-lived transactions due to the fact that much work might have been done and will be lost in the event of a failure. In addition, long-lived transactions typically access many data items during the course of its execution. Due to the isolation requirement of the traditional transaction model, these data items cannot be released until the transaction commits. Long-lived transactions run for hours and sometimes days thus causing short transactions to wait for long periods of time to access those data items accessed by long-lived transactions.

Another limitation of the traditional transaction model is that it does not allow much cooperation among activities. Many applications require cooperation among activities. For example, in CAD environments several people may be jointly working on a project where each person is responsible for part of the design project. Thus cooperation is needed in order to complete the project and is usually achieved through shared data items; specifically, the data items are accessed *alternately* by the people working on the design project. This form of cooperation is prohibited in the traditional transaction model due to the isolation requirement of uncommitted transaction results.

Another problem with the traditional transaction model is that it may be too strict for certain environments. As an example, in a workflow application, some of the (sub)tasks that deal with invoices may have to satisfy the ACID properties (on a small portion of the database) whereas other tasks may work on their own copy of the data objects and only require synchronization. As another example, in design environments, compensating actions (or even partial rollbacks) may be more appropriate when a long running design activity reaches an undesirable point than aborting or rolling back a transaction to its starting point.

In the following section we give two examples of applications not well-served by the ACID properties provided by the traditional transaction model. These examples are included to better illustrate the limitations of the traditional transaction model.

1.6 Applications Not Served by ACID Transactions

Let us consider two applications which are not well-served by the traditional transaction model, specifically, a trip planning application and bulk updates (as an example of a long-lived application). Details of these two applications along with how the traditional transaction model does not adequately serve their requirements, are given below.

Trip planning. Assume one needs to travel from Orlando, Florida, to Alexandria, Egypt. Since there are no direct flights between these two cities, several connecting flights need to be booked along with possible car rental reservations, train or bus reservations, or hotel room reservations for overnight stays. Furthermore, assume that the traveler has certain preferences regarding the trip; specifically, the traveler does not prefer driving during the night and would rather stay overnight at a hotel. In addition, the traveler prefers to book as many flights on the same day as possible. Given this information the travel agent comes up with an itinerary for the trip. The travel agent, using a transaction-based reservation system which gives access to all the required airline databases, train and bus companies, and hotel databases, comes up with the following itinerary:

STEP 1: book flight from Orlando, Florida, to Detroit, Michigan

STEP 2: book flight from Detroit, Michigan, to Amsterdam, Netherlands

STEP 3: book flight from Amsterdam, Netherlands, to Cairo, Egypt

Assume that after completing step three, the travel agent realizes that the only method to get from Cairo to Alexandria is by renting a car. Now, this poses a problem since the traveler does not prefer to drive during the night. What can the travel agent do in this situation? One solution may be to book a flight from Detroit to Mansoura, Egypt, and then reserve a train from Mansoura to Alexandria. Another solution is to book a flight from Detroit, Michigan, to New York and then take a direct flight from New York to Alexandria. Therefore, given the flat structure of the traditional transaction model, the travel agent has only two choices: i) issue a rollback operation after execution of step 3. This unnecessarily wastes a lot of previous work since there is no need to cancel the reservation from Orlando to Detroit, or ii) explicitly cancel those reservations that are no longer useful and keep those reservations which can still be used. This latter approach may be cumbersome if the amount of work to be undone is much larger than that shown in the example above (e.g., if a group of people were traveling on the same trip together).

From the above example, it is apparent that rather than only having the total rollback provided by the traditional transaction model, a transaction model should also provide a *selective* rollback. In other words, rather than aborting the entire transaction and losing some useful work, the user should be able to rollback to a particular position in the transaction body, in this case to step 2.

Bulk updates. Another example which illustrates the limitations of the traditional transaction model is bulk updates. Assume that the same operation is to be performed repeatedly on different data items. In addition assume that the number of data items is very large. A typical example of this is in credit card agencies where the accumulated interest is computed at the end of each month. Let us assume that there are a million credit card accounts which need to be updated.

STEP 1: read credit card balance

STEP 2: balance = balance * (1 + interest_rate)

STEP 3: Total-Credit-Balance = Total-Credit-Balance + balance

Let us assume that the above transaction begins execution, and after the successful update of 955,843 credit card balances, a failure occurs. Now, since the traditional transaction model dictates that a transaction should be *one atomic unit*, all these accounts need to be changed to their original values. This wastes a very large amount of work in terms of both updating the accounts, to reflect the interest incurred, as well as reversing the accounts back to their original value. It may be argued that instead of creating only one transaction to update all accounts, each account should be updated by a stand-alone transaction. This argument aims at reducing the amount of work lost when a crash occurs since at most only one account balance needs to be reversed. This solution however is incorrect since the semantics of the operation is to update all accounts at the end of the month. By adopting this solution the system will be unable to update all accounts in the event of a crash simply due to the fact that it will have no information about which account was last updated. Consequently, the system will be unable to pick up the work after recovery.

This dissertation is structured as follows. Chapter 2 presents the current solution adopted by the database community for overcoming the limitations of the traditional transaction model. The solution basically proposes extended or advanced transaction models that better serve the requirements of non-traditional applications. Chapter 2 also describes the problem which we address in this dissertation, namely, *how* to provide an *extensible transaction management facility*, specifically, how to support different transaction models on the same underlying DBMS. Motivation for this work is also given is chapter 2. Chapter 3 describes a number of extended transaction models. A table summarizing the structure and correctness of these advanced transaction models is also included in chapter 3. A detailed survey of related work, namely, different approaches to supporting various transaction models is presented in chapter 4. Chapter 5 then describes the notion of the active database paradigm and examines its adequacy for realizing the semantics of various transaction models. Chapter 6 then outlines our approach based on the active database paradigm as well as presents details of several alternative ways for supporting extended transaction models. The implementation of our approach uses *Sentinel*, an active OODBMS developed at UF, as the underlying platform. A brief overview of Sentinel and *its* adequacy for supporting various transaction models is given in chapter 6 as well. In addition, the alternatives for supporting transaction models in an OO environment are given. Our implementation details along with the ECA rules which formulate the semantics of nested transactions, Split transactions and Sagas are presented in chapter 7. Chapter 8 discusses the extensibility of our approach while our conclusions and future directions are given in chapter 9.

CHAPTER 2
PROBLEM STATEMENT AND MOTIVATION

The current solution to meeting the diverse requirements of non-traditional applications has been the proposal of a number of advanced or extended transaction models such as nested transactions, Sagas, ConTract model, and Flex transaction model [35, 23, 41, 21]. These transaction models relax the ACID properties in various ways to better model the parallelism, consistency, and serializability requirements of non-traditional applications. Proponents of advanced transaction models primarily start from a specific application. An application's dynamic behavior is analyzed, a fault model is specified, and features are either added or modified to the classical ACID transaction model aiming at supporting the requirements of that application. For instance, enforcing the isolation and failure atomicity properties may be inadequate for long-running transactions; short transactions may be forced to wait for long periods due to the isolation property, and failure atomicity dictates rolling back large amounts of work in the event of failures. Sagas [23] relax the isolation and consistency properties of the traditional transaction model thereby better modeling long-lived transactions.

The approach of rolling new variants of transactions as applications emerge is unlikely to provide a realistic solution to the general problem primarily for three reasons: i) an increase in the number of new applications having varying transaction requirements leads to a proliferation of transaction models which are required for their support, ii) this proliferation increases the difficulty of determining whether an application's processing requirements are served by extant transaction models, and iii) this proliferation increases the difficulty of integrating the various models in a uniform manner into a DBMS.

Currently, a transaction model (traditional or otherwise) is typically hardwired into a DBMS. Since *no one* transaction model satisfies the requirements of *all* classes of applications, choosing a transaction model at system implementation time clearly limits the applications that can be supported. Several approaches to overcome this problem have been proposed:

- Object services architecture (OSA) is a software architecture consisting of a collection of independent (orthogonal) software services, all of which operate via a software backplane or message passing bus [10]. TI's Open OODB prototype has taken this approach for supporting various services [47, 37] but have *not* addressed the transaction model issues.

- Carnot [3] has taken the approach of providing a general specification facility that enables the formalization of most of the proposed transaction models that can be stated in terms of dependencies amongst significant events in different subtransactions. CTL (Computational Tree Logic) is used for the specification and an actor based implementation has been used for implementing task dependencies.

- ASSET [6] identifies a set of primitives using which a number of extended transaction models can be realized. Implementation of the primitives has only been sketched.

- ACTA [18] proposed a framework for specifying, analyzing, and synthesizing extended transaction models using dependencies.

- A proposal for supporting advanced transaction models by extending current transaction monitors' capability [33].

The approaches taken so far for supporting extended transaction models can be broadly classified into i) identifying a set of primitives with which a number of extended transaction models can be realized, and ii) providing a general specification language which allows the specification of dependencies among transactions. The first approach examines extant transaction models and identifies *a set* of transaction primitives–such as begin, commit, abort, delegate–capable of expressing these transaction models. The drawbacks of this approach are the following: i) lacks extensibility as it is highly likely that with the emergence of newer transaction models the semantics of existing primitives may need to be modified or new primitives introduced,[1] ii) adopts a procedural approach for specifying transaction models thereby *embedding* transaction semantics inside applications. This poses two main problems, namely, it requires existing applications to be rewritten to achieve the desired transaction semantics and it increases the difficulty of changing the transaction semantics of applications. More importantly, embedding transaction semantics in applications renders it impossible to *dynamically* modify the control flow. Dynamic modification of control flow is necessary in many workflow applications, and iii) offers coarse-grained control since a user can express transaction semantics at high levels such as begin and commit; the user cannot express semantics at low levels such as the object level, transaction table level and log level. The second approach takes the view that various dependencies (such as commit and abort dependencies) exist between transactions and that providing a framework for expressing and enforcing dependencies is sufficient for describing any transaction model. The primary limitation of this approach is in providing a framework general enough to express existing as well as new types of dependencies (as they become available). Furthermore, little attention has been paid to allowing transaction dependencies to be modified, added, or deleted dynamically.

We take the view that the two approaches outlined above do not offer the flexibility and expressive power for modeling existing extended transaction models, newer transaction models as they become available, and arbitrary transaction semantics. This is primarily because the approaches only offer coarse-grained control; transaction semantics can only be

[1]As an example, to model the Split transaction model, the *split* and *join* transaction primitives are required.

expressed in terms of *high-level operations* (such as commit and abort) and *dependencies* among transactions (such as commit dependencies). Furthermore, the transaction semantics are embedded within application code making it difficult to change as well as impossible to modify dynamically. Our research was prompted by the limitations of extant approaches, namely,

- Current approaches support extended transaction models by examining the requirements of *existing* transaction models. This produces an inextensible system as new transaction models or variations of existing transaction models are likely to emerge.

- Existing approaches *embed* the transaction semantics in the application code itself. This does not offer a clean separation between transaction semantics and the application code, a feature necessary for easily modifying the transaction semantics and application code independently of each other.

- Very little attention has been paid to understanding the *interactions* of applications whose transaction semantics differ; specifically, can applications using different transaction models run concurrently on a DBMS? What properties (e.g., serializability) must be enforced between these transactions? What are the implications of running different transaction models concurrently on performance, deadlock detection algorithms, recovery etc.?

- Although several efforts for supporting transaction models have been proposed, the implementation details for supporting multiple transaction models have been sketched; even concrete prototypes are lacking.

CHAPTER 3
ADVANCED TRANSACTION MODELS

In this chapter we give a detailed survey of a number of proposed advanced transaction models. The transaction models which are described here include nested transactions, Sagas, ConTract, Flex, Cooperative Transaction Hierarchy and Cooperative Software Engineering Environments. The characteristics and advantages of each transaction model are given along with the application domains they serve. Furthermore, a table, taken from [22], summarizing the characteristics of these transaction models is given at the end of the chapter.

3.1 Nested Transactions

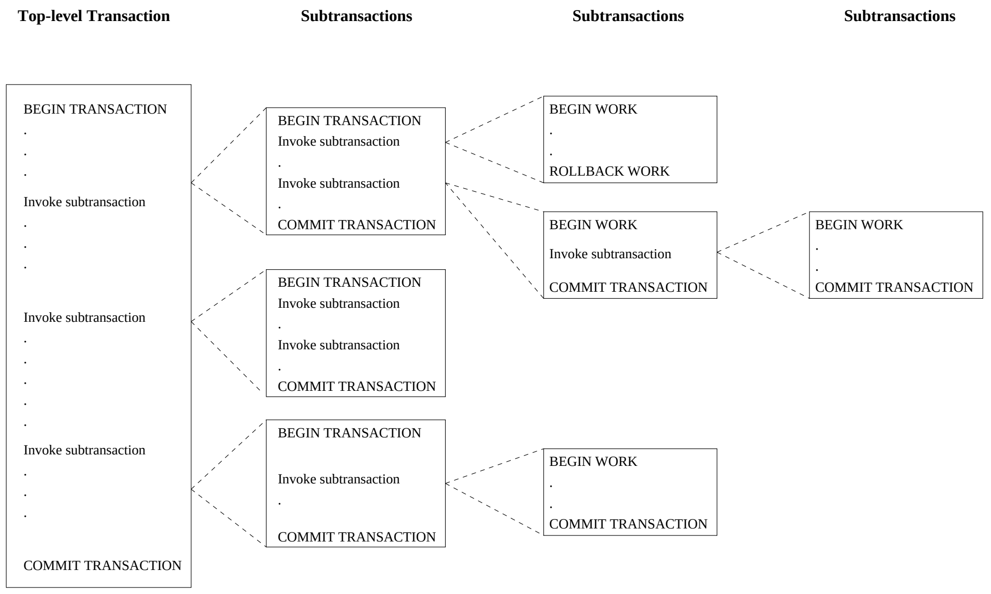


Figure 3.1. Structure of Nested Transactions.

The traditional transaction model is found to be inadequate when serving the requirements of complex applications such as object oriented systems, long-lived transactions, or distributed systems. Nested transactions [35] were proposed to overcome these limitations and better serve the requirements of such systems. In the nested transaction model, the notion that transactions have a flat structure is extended; specifically a transaction may contain any number of subtransactions, and each subtransaction, in turn, may contain any number of subtransactions. Hence, the entire transaction forms a hierarchy of transactions the root of which is called the *root* or *top-level* transaction. An example of the structure of nested transactions is depicted in Figure 3.1. Transactions that have subtransactions are called *parents*, and their subtransactions are termed their *children*. The transactions on the path from a transaction to the root of the transaction tree are called the *superiors* of the transaction.

With respect to transaction semantics, top-level transactions have all the properties of traditional transactions; that is, they preserve the ACID properties. Nested transactions preserve serializability among subtransactions; therefore, subtransactions cannot cooperate or share data. The commit of a subtransaction is conditionally subject to the commit of its superiors. Hence, a subtransaction's updates become permanent only when the enclosing top-level transaction commits. Upon commit of a subtransaction, all locks held are inherited by the parent transaction. A parent transaction does not interfere with its children (in sibling concurrency); a transaction is allowed to hold a lock if the conflicting transaction is one of its superiors.

There are two main advantages of the nested transaction model over traditional transactions, namely increase potential for parallelism and finer control over failures. Since the nested transaction model allows subtransactions within a top-level transaction to execute concurrently, intra-parallelism is achieved. Moreover, since subtransactions belonging to different top-level transactions may also execute in parallel, inter-transaction parallelism is also achieved. In the traditional transaction model, if a failure occurs, the entire transaction is rolled back to ensure failure atomicity. This is in contrast to the nested transaction model where, if a subtransaction fails, the user has the flexibility to retry the subtransaction or abort the entire transaction. The finer control over failures is the property which makes the nested transaction model highly desirable for distributed applications and applications with long-running activities where the probability of failures and faults is high.

3.2 Sagas

Sagas [25] is a transaction model introduced to adequately serve the requirements of long-lived transactions. The concept of Sagas is based on the notion of *compensating transactions*. A Saga consists of a set of independent component transactions $T_1, T_2, ..., T_n$ where each component transaction T_i (except transaction T_n) has an associated compensating transaction CT_i. Compensating transactions *semantically undo* the effects of their respective component transaction.

The transactions $T_1, T_2, ..., T_n$ execute serially in a predefined order and may interleave arbitrarily with the component transactions of other sagas. If a component transaction aborts, then the entire Saga aborts by executing the compensating transactions in reverse order to the order of the commitment of the component transactions. Since the component transactions of a saga may arbitrarily interleave with the component transactions of other sagas, consistency is compromised. Furthermore, once a component transaction completes execution, it is allowed to commit and release its partial results to other transactions thereby relaxing the isolation property. However, Sagas preserve both the atomicity and durability properties.

Several variants of Sagas have been proposed. One variant requires serial execution of component transactions while other variants allow concurrent execution of component transactions. Other variants adopt a forward recovery policy where the remaining component transactions are executed in the event of a component transaction aborting while other variants adopt a backward recovery policy of executing the compensating transactions of all the already committed transactions.

3.3 Split Transactions

Split transactions [40] were proposed mainly for supporting open-ended applications. In this transaction model, a transaction can execute the operation *split-transaction* which basically creates a *new* top-level transaction. The original transaction and the new transaction are serializable. When the original transaction executes the operation *split-transaction*, it can delegate responsibility of uncommitted operations on a particular set of objects to the newly created transaction. After the split occurs, the two transactions continue execution and commit or abort independently. Similarly, a transaction can also execute the operation *join-transaction* which essentially combines two active serializable transactions into one transaction. The main advantage of split transactions is relaxing isolation, which is achieved when either the original or new transaction commits and releases its results.

3.4 Cooperative Transaction Hierarchy

The cooperative transaction hierarchy [36] concept was introduced to overcome the strict correctness criteria dictated by the serializability requirement of the traditional transaction model. In this transaction model, cooperation among transactions is allowed and is achieved by allowing the substitution of user-defined correctness for the notion of correctness defined by serializability. Basically, a cooperative application is structured as a rooted tree referred to as a *cooperative transaction hierarchy* where the external nodes (leaf nodes) represent the transactions associated with the individual designers. An internal node (a node which has at least one child) is referred to as a *transaction group* while all nodes having the same parent are referred to as *cooperative transactions.* The execution of cooperative transactions need not be serializable; rather, the transaction group (i.e., the parent) defines a set of rules that regulate interactions among the children, i.e., the cooperative transactions. Finite-state automata is used to specify the rules, where each finite-state automata defines, for a particular set of objects, the operations allowable for each cooperative transaction on that set of objects and the valid ways of interleaving the operations of related cooperative transactions.

In summary, the main contribution of the cooperative transaction hierarchy concept is the relaxation of the strict serializability requirement of the traditional transaction model. In this model, it is possible for the user to define his/her own notion of correctness and thus allow cooperation among tasks. The allowance of user-defined correctness relaxes the isolation requirement thereby helping in alleviating the problems caused by long-lived transactions.

3.5 Cooperative SEE Transactions

Cooperative SEE transactions [27] were proposed to support the transaction requirements needed in software engineering environments. In such environments, support for long-lived transactions, cooperative transactions, as well various levels of cooperation, is needed. The flexibility of supporting various degrees of cooperation is needed since the level of cooperation is typically dependent on the particular application or environment.

In this transaction model, there is no concept of the transaction manager having a *single* built-in notion of correctness; rather, this transaction model supports a transaction manager which is programmed to accept application-specific protocols, where these protocols enforce the appropriate correctness required by the SEE. The protocols can be used to describe:

- **Conflicts.** Operations that are not allowed to execute concurrently. For example, no member of group G1 may edit a piece of code which is currently being edited by a member of group G2.

- **Patterns.** Sequences of requests which must occur before a transaction can commit or a request accepted for execution. This can be viewed as a predicate which must be satisfied before a transaction can commit. For example, after linking the system, a set of test suites needs to be executed before releasing the new version of the code.

Therefore, after linking the system, all execution requests should be rejected until the test suites have been executed and yield favorable results.

- **Triggers.** Actions that are taken when a request begins or ends. For instance, it is necessary to make a copy of a system module prior to updating it. This is necessary in order to isolate the released version from the experimental version.

- **Commit or abort semantics.** Actions to be taken when a transaction ends. For example, after updating a system module and testing its correctness, it is necessary to release the new version for public access.

3.6 ConTract Model

The ConTract transaction model [42] was proposed to provide a generalized control mechanism for long-lived activities. Transactions in the traditional transaction model perform small units of work, access a few data items and thus have a short duration. Furthermore, transactions in the traditional transaction model are viewed as concurrent and completely unrelated units of work. Consequently, existing interrelations between individual transactions, like control flow dependencies and other semantic connections, cannot be implemented by the system. Rather, these dependencies and other connections need to be handled by the application. For example, consider the following specification of a simple transaction sequence: *Execute transaction T1. Upon successful completion of transaction T1, execute transactions T2, T3 and T4 concurrently. After transactions T2, T3 and T4 commit, start execution of transaction T5. In the event of the abort of any one of transactions T2, T3 and T4, abort the other two transactions as well as rolling back transaction T1.*

The above specification cannot be modeled in the context of the traditional transaction model without the introduction of further application programming. The basic idea behind the ConTract model is to build large applications from short ACID transactions and to provide an *application independent* system service, which exercises control over them. Therefore, the burden of tasks such as controlling parallel or concurrent computations and scheduling distributed or non-distributed executions are removed from the application programmer.

A ConTract is basically a consistent and fault tolerant execution of an arbitrary sequence of predefined actions referred to as *steps* according to an explicitly specified control flow description referred to as a *script*. Steps are the elementary units of work in the ConTract model where each step implements one basic computation of an application, e.g., debiting a bank account, booking a flight or reserving a hotel room. No internal parallelism exists within a step and thus each step can be coded in an arbitrary sequential programming language. With regard to the size of a step, it is determined by the amount of work an application can tolerate to be lost after a system failure. A script, on the other hand describes the control flow and all other execution dependencies of a long-lived activity. Control flow can be modeled by sequence, branch, loop and some parallel constructors. Therefore, a ConTract is a program which has control flow like any other programming environment, that has persistent local variables, accesses shared objects with application oriented synchronization mechanisms, and which has precise error semantics.

As previously mentioned, scripts describe the structure or the control flow of a complex activity while the steps implement the algorithmic parts. A *ConTract Manager* is introduced to handle all aspects concerning execution control at run time. Basically, a ConTract manager uses a predicate transaction to specify activation and termination conditions for a step. A step is executed if the predicate for its activation evaluates to true and the required execution resources are available. Similarly, after each step terminates a set of conditions are evaluated. Each condition which evaluates to true triggers the execution of one or more steps.

3.7 Flex Transaction Model

The Flex transaction model [20] was proposed to allow more flexibility in transaction processing. In this transaction model, a transaction is viewed as a set of tasks and the user is allowed to specify, for each task, a set of *functionally equivalent* subtransactions, each of which when completed will accomplish the task. The execution of a Flex transaction succeeds if all of its tasks are accomplished. The failure property of the traditional transaction model is relaxed since a Flex transaction may proceed and commit even if some of its subtransactions fail. Furthermore, dependencies such as *failure dependencies, success-dependencies and external-dependencies* can be specified on the subtransactions of a Flex transaction. Failure dependencies and success dependencies define the execution order on the subtransactions. On the other hand, external-dependencies define the dependencies of the subtransaction execution on the events that do not belong to the transaction. The isolation requirement of the traditional transaction model is relaxed in the Flex transaction model. This is accomplished by allowing the user to define and use compensating transactions. All of the above-listed features contribute to the flexibility of the Flex transaction model, which is useful for transaction processing in multidatabase systems. In summary, the Flex transaction model is a generalization of traditional transactions and has the following features:

- **Function replication.** Certain tasks may be accomplished by more than one local system. For example, on a trip, the automobile rental agency where the car is rented from may be unimportant. Therefore, the transaction programmer may leave the system the choice of renting from Budget, Hertz, National etc. This property is referred to as function replication and is incorporated in Flex transactions to allow flexible composition of global transactions.

- **Mixed transaction.** Some transactions can be *semantically undone* even after they are completed. For example, consider booking a seat on a flight from New York to

California. After a subtransaction reserves the seat on this flight it can be compensated by another subtransaction which cancels this reservation. The fact that some transactions can be semantically undone allows subtransactions to commit before the corresponding global transaction has committed. Global transactions consisting of both compensatable and non-compensatable subtransactions are referred to as *mixed transactions.*

- **Value function.** In the Flex transaction model, each subtransaction and global transaction may have associated with it a *value function.* A value function denotes the time of completion of each transaction and is used to ensure order correctness. For example, the reservation of a seat for a flight is made before the trip.

The formal definition of Flex transactions is [22]

A Flex transaction T is a 5-tuple (B, S, F,Π, f) where

- B = $t_1(type_1), t_2(type_2), ..., t_n(type_n)$ is a set of typed subtransactions. In particular, B specifies whether a transaction t_i is compensatable ($type_i$ = "C") or not ($type_i$ = "NC"). A compensatable transaction may commit earlier, but its effects may be later "semantically undone".

- S is a partial order of B, called the success order of T. $t_i < t_j \in$ S means that transaction t_i has to be successfully executed before transaction t_j may be started. S describes internal dependencies.

- F is a partial order of B, called the failure order of T. F is analogously defined to S. If $t_i < t_j \in$ F, then transaction t_j depends on the failure of transaction t_i. F also describes internal dependencies.

- Π is a set of external predicates on B. Π may contain cost functions, time constraints (e.g., when a local transaction may be scheduled), etc.

- f is an n-ary boolean function and is called the acceptability function of T. f describes *all* acceptable states. For example, f$(x_1, x_2, x_3, x_4) = (x_1 \wedge x_3) \vee (x_2 \wedge x_4)$ is defined over 4 transactions on local databases and states that the Flex transaction can be accepted when either transactions 1 and 3 or transactions 2 and 4 are a success. $x_1, ..., x_4$ are the execution states of $t_1, ..., t_4$. The subset C $\subseteq$ B of all subtransactions which finally are committed is called the commit-set of T. C corresponds to one acceptable state.

3.8 Polytransactions

Multiple databases in large companies are required to serve the needs of various application systems. A major concern in such environments is the management of these databases, specifically, maintaining the consistency of inter-related data. The term *interdependent data* is used to denote the existence of an integrity constraint between two or more data items residing in different databases. The integrity constraint specifies the data dependency and the consistency requirements between these data items. Management of such data implies that a certain degree of mutual consistency among interdependent data is maintained. Therefore, the manipulation, including concurrent updates, of the interdependent data must be controlled.

Polytransactions [43] were developed with the goal of finding a transaction model that would facilitate the support of interdependent data in mulitdatabase environments. Two important features of this model are i) declarative specification of interdependent data and their mutual consistency requirements, and ii) use of the specification to automatically generate related update transactions that manage interdependent data. The separation of the constraints from the application programs facilitates the maintenance of data consistency requirements and allows flexibility in their implementation. It also allows investigation of various mechanisms for enforcing constraints, independently of the application programs. Thus, changes in the application programs can be made independently of the constraints, and vice versa. By grouping the constraints together, we can check their correctness and discover possible contradictions among them.

In this model, Data Dependency Descriptors (D^3) are used to specify interdatabase dependencies. Each D^3 is a 5-tuple:

$$D^3 = < \text{S, U, P, C, A} > \text{ where}$$

- S is the set of *source data objects*
- U is the *target data object*
- P is a boolean valued predicate called *interdatabase dependency predicate* (dependency component). It specifies a relationship between the source and target data objects, and evaluates to true if this relationship is satisfied.
- C is a boolean-valued predicate, called *mutually consistency predicate* (consistency component). It specifies consistency requirements and defines when P must be satisfied.
- A is a collection of *consistency restoration procedures* (action component). Each procedure, specifies actions that must be taken to restore consistency and to ensure that P is satisfied.

A polytransaction (T^+) is a *transitive closure* of a transaction T submitted to an interdependent data management system. A ploytransaction can be represented by a tree in which the nodes correspond to its component transactions and the edges define the *coupling* between the parent and children transactions. Given a transaction T, the tree representing its polytransaction T^+ can be determined as follows. For every data dependency descriptor D^3, such that the data item being updated by T is among the source objects of the D^3, we look at the dependency and consistency predicates P and C. If they are satisfied, no further transactions will be scheduled. If they are violated, we create a new node corresponding to a system generated new transaction T' (child of T) to update the target object of the D^3. T' will restore the consistency of the target object, so that the P and C predicates will be satisfied. Specification of weaker mutual consistency criteria will result in infrequent violations of C. Hence the restoration procedures (and the corresponding child transaction) will be scheduled less often.

3.9 S Transaction Model

The semantic transaction model [19], commonly referred to as the S transaction model, evolved from the hierarchical transaction models. It was first developed for a large-scale inter-organizational autonomous environment, international banking. In this environment cooperation and local autonomy are of a major concern.

S transactions basically have the same structure as nested transactions; i.e., they also form a hierarchy of transactions. However, the semantics of S transactions differ considerably from that of nested transactions, primarily in three ways. First, the recovery mechanism adopted in the nested transaction model is based on the concept of rolling back the work when a failure occurs. In the S transaction model, however, recovery is based on the notion of compensating transactions; specifically, each transaction can be undone by executing a corresponding compensating transaction which semantically undoes the transaction. The second difference between these two models is related to the isolation property. In the nested transaction model, when a subtransaction completes its execution, it does not commit and release its results to the outside world. Instead, the results are released only to the superiors of the subtransaction. This differs from the S transaction model where when a subtransaction completes its execution, it commits thereby releasing its results to the outside world. Therefore, in the S transaction model isolation is preserved only at the subtransaction level. Third, due to the local autonomy requirement in the S transaction model, a component system issuing an S transaction cannot impose on other systems which additional systems should participate in the execution of the S transaction. Each component system may execute a subtransaction any way it chooses. Upon failure of a subtransaction, the component system may issue a request to another system to execute that subtransaction. This feature naturally causes the execution tree of an S transaction to be indeterminate and therefore may vary from one execution to the other.

Finally, in S transactions, the local transactions are the concurrency control units, and the recovery unit can be dynamically chosen to be any sub-S-transaction. Both backward and forward recovery are supported where backward recovery is based on the conventional abort mechanism or semantically undone using compensating transactions.

Model	Transaction Structure	Subtransaction Types	Transaction Correctness	Comments
Traditional	-	-	ACID	
Nested	Subtransaction Hierarchy	Contingency, Non-Vital	ACID	
Sagas	Subtransactions	Compensating	ACID^{C}	
Split	Dynamic Restructuring (Split and Join)	-	Original Semantics Preserved	Supports long-lived and cooperating transactions
ConTract	Subtransactions	Compensating, Contingency, Non-Vital	ACID^{C}	Supports long-lived and cooperating transactions
Flex	Subtransaction Hierarchy	Compensating, Contingency, Non-Vital	ACID^{C}	Supports long-lived and cooperating transactions
S	Subtransaction Hierarchy	Compensating, Contingency, Non-Vital	CD to ACID^{AC}	For international banking and general MDBSs
Cooperative	Subtransaction Hierarchy	Compensating	User-Defined	For cooperative environments, e.g. design databases
Cooperative SEE	Subtransaction Hierarchy	Trigger	User-Defined	Cooperative work, e.g., engineering environments
Polytrans.	Dynamic Transaction Hierarchy	-	User-Defined	Supports interdependent data

^{C} For transactions where subtransactions are compensatable.

^{A} Depending on the degree of component system autonomy.

Figure 3.2. Characteristics of Advanced Transaction Models.

CHAPTER 4
RELATED WORK

Several efforts [5, 24] have proposed approaches for supporting extended transaction models. In this section we provide a detailed examination of these efforts with special emphasis on the specification of extended transaction models and their support, the expressiveness of the system in terms of transaction models supported, and the extensibility of the system.

ASSET [5] provide a flexible transaction facility that allows users to define customized transaction semantics in applications. ASSET consists of a set of transaction primitives which are classified into basic and new primitives. Basic primitives are similar to those found in most transaction processing systems and are *initiate(f, args), begin(t), commit(t), wait(t), abort(t), self()* and *parent()*. The new primitives, *delegate*(t_i, t_j, *ob-set), permit*(t_i, t_j, *ob-set, operations)* and *form-dependency(type,* t_i, t_j*)*, permit the construction of arbitrary transaction models and the realization of relaxed correctness criteria. It is important to note that these transaction primitives are not expected to be directly used by the user. Rather a high-level description of the required transaction model is specified by the user which is subsequently translated into code which uses these primitives. Below, we provide a brief description of these primitives.

Briefly, initiate(f, args) registers a new transaction that executes the function f with the arguments *args*. The primitives begin(t), commit(t) and abort(t) respectively start, commit and abort the transaction whose tid is t. The commit primitive is a blocking primitive in the sense that if it is issued before a transaction completes execution, it will wait until the execution completes. The self() and parent() primitives each return a transaction tid; the former returns the tid of the executing transaction and the latter the tid of the executing transaction's parent. Waiting for a transaction t to complete is accomplished by using wait(t) which returns the value 1 when transaction t commits and returns the value 0 if t aborts. The primitive delegate(t_i, t_j, ob-set) transfers the responsibility of operations performed on *ob-set* by transaction t_i to transaction t_j, i.e., these operations are committed only if transaction t_j commits (unless transaction t_j delegates them to another transaction). Cooperation amongst transactions is achieved by using the permit primitive; permit(t_i, t_j, ob-set, operations) means that transaction t_i permits transaction t_j to perform conflicting *operations* on objects in *ob-set* without conceptually creating a conflict edge in the serialization graph from t_i to t_j. The semantics of this operation is i) t_j is now allowed to execute *operations* on objects in *ob-set* without having to wait, ii) only one transaction, i.e., t_i or t_j, can perform an update operation at any given time on the same object, and ii) once transaction t_i permits t_j to perform an operation on an object, t_j can permit other transactions to perform operations on that object. The last primitive form-dependency(type, t_i, t_j) establishes a dependency of the specified *type* between t_i and t_j, where *type* includes commit, abort and group commit dependencies. A commit dependency (CD) between transactions t_i and t_j, denoted by form-dependency(CD, t_i, t_j), implies that if both t_i and t_j commit, then t_j *cannot commit before* t_i commits. However, if t_i aborts, then t_j can still commit. The abort dependency (AD), denoted by form-dependency(AD, t_i, t_j), implies that if t_i aborts then t_j must also abort. The group commit dependency (GC), denoted by form-dependency(GC, t_i, t_j), means an all or nothing option, i.e., either both t_i and t_j commit or neither commit.

To illustrate how these primitives are used assume that trip arrangements are being made and it involves reserving both an airline ticket and a hotel room. If either of these reservations cannot be made, then the trip should be canceled. This can be easily specified using the nested transaction model. First, the user defines a high-level specification of the nested transaction which is:

tid t;

t = trans {}

trans { make-airline-reservation();}

trans { make-hotel-reservation();}

The above specification is then translated by a compiler into :

tid t;

t = initiate(trip);

begin (t);

commit(t);

where function trip is synthesized as follows:

```
void trip()

{

tid t1;

t1 = initiate(make-airline-reservation);

permit(self(), t1); /* transaction t allows t1 to access it's objects */

begin(t1);

if(!wait(t1))

          abort(self());

delegate(t1,self());

commit(t1);
tid t2;

t2 = initiate(make-hotel-reservation);

permit(self(), t2); /* transaction t allows t1 to access it's objects */

begin(t2);

if(!wait(t2))

          abort(self());

delegate(t2, self());

commit(t2);

}
```

The data structures and the algorithms used to implement these primitives were described in a modified version of the EOS storage manager. The main data structures used are a transaction descriptor table, an object description table and a transaction dependency graph. The transaction descriptor table is a hash table where each entry is a transaction descriptor which maintains information about a transaction.

The transaction dependency graph is a directed graph where the nodes represent transactions and the arcs represent the type of dependency between two nodes. For example, an arc of type commit from transaction t_i to transaction t_j denotes a commit dependency between the two transactions. For a detailed discussion of how the primitives and these data structures interact we refer the reader to ASSET [5].

Comments

1. The high-level language constructs provided for specifying transaction models are not sufficient for defining all transaction models. Constructs for popular transaction models like cooperative, nested and sagas are provided. However, no constructs are provided for transaction models such as nested transactions with only sibling concurrency and nested transactions with parent and child concurrency only[1]. This

[1]These are variations of the nested transaction model

compromises system extensibility since the introduction of newer transaction models or proposal of variations of existing ones requires changing or adding constructs. In addition, there might be even a need to change or add transaction primitives.

2. A procedural approach is taken for supporting extended transaction models. The primary disadvantage of this is the need for developing a language which is general enough to specify existing as well as new transaction models. Furthermore, a compiler for the language needs to be developed and possibly modified as new constructs are added to the language.

3. The above approach has the disadvantage of limiting concurrency, thereby decreasing transaction throughput. To see this, consider modeling the nested transaction model where only sibling concurrency is allowed, i.e., the parent must suspend its execution until its children complete execution. Using the trip example mentioned above, the code (using the primitives provided) accomplishing that is

   tid t;

   t = initiate(trip);

   begin (t);

   commit(t);

   where function trip is synthesized as follows:

```
void trip()
{
tid t1, t2;
t1 = initiate(make-airline-reservation);
t2 = initiate(make-hotel-reservation);
permit(self(), t1); /* transaction t allows t1 to access it's objects */
permit(self(), t2); /* transaction t allows t1 to access it's objects */


begin(t1);

begin(t2);


/* now the siblings t1 and t2 are executing concurrently */


if(!wait(t1))   /* LINE 12 */

abort(self());

delegate(t1,self());

commit(t1);


if(!wait(t2))

abort(self());

delegate(t2, self());

commit(t2);

}
```

4. In the section describing the data structures necessary for implementing ASSET, they refer to a pending lock list which never seems to be used.

   Now, at the 12th line transaction t is waiting for t1 to complete. While t is waiting, it is possible that transaction t2 aborts in which case, the entire nested transaction should be rolled back and the resources released. However, since a programmatic/procedural approach is adopted, transaction t will remain waiting for transaction t1's outcome, possibly unnecessarily.

5. They claim that they introduce the three primitives delegate, permit and form-dependency. However, these were previously introduced in ACTA.

TSME [24] provide a transaction processing system toolkit, referred to as a Transaction Specification and Management Environment (TSME), which supports the definition and construction of application-specific extended transaction models (ETMs). The TSME is discussed in the context of a Distributed Object Management System (DOMS) which is an object-oriented environment allowing the integration of autonomous and heterogeneous local systems and the development of non-traditional applications.

The three main components of the TSME are a transaction specification language, a Transaction Dependency Specification Facility (TDSF), and a programmable transaction management mechanism (PTMM). The transaction specification language is used by transaction model designers for writing *implementation-independent* specifications of extended transactions. These specifications are subsequently submitted to the TDSF for acceptance or rejection. Once an ETM specification is accepted by the TDSF, it is stored in the TDSF repository and the corresponding ETM is supported by the TSME. The last component of the TSME, the PTMM, supports the implementation of ETMs by configuring a run-time transaction management mechanism (TMM) to ensure the preservation of transaction dependencies.

In TSME [24] extended transactions are viewed as complex transactions; complex transactions consist of a set of constituent transactions and a set of dependencies between them. Each constituent transaction can in turn be either simple or complex. A complex transaction T is denoted by T = (T_T, T_D) where T_T is the set of constituent transactions and T_D are the set of dependencies between them. Each complex transaction T that is not a constituent transaction of any other transaction has the following two kinds of transaction dependencies:

- intra-transaction dependencies–these define the relationships between the constituent transaction of T

- Inter-transaction dependencies–these define the relationships between T and all transactions that are not constituent transactions of T

Transaction dependencies are specified using 5 tuple elements of the form (T_i, τ, O, E_n, Post), where T_i is the dependent transaction, τ is the set of transactions that T_i depends on, O is the set of objects that the dependency must consider, and E_n and Post are logical predicates representing the enabling condition and the postcondition, respectively. E_n denotes when the postcondition must be evaluated while evaluation of the postcondition determines whether the dependency is satisfied or not.

Intra-transaction dependencies are further classified into *transaction state dependencies* and *correctness dependencies.* Transaction state dependencies express relationships between the states of transactions where a transaction can be in either the begin, prepare, commit or abort state. Three kinds of state dependencies: *backward, forward* and *strong* are supported. A backward state dependency between a pair of transactions T_i and T_j, denoted by (T_i, T_j, O, T_i.state = X, Y(T_j) < X(T_i)), means that T_i cannot enter state X *before* T_j has entered state Y. A forward state dependency denoted by (T_i, T_j, O, T_i.state = X, $\neg(Y(T_j) < X(T_i))$), means that T_i cannot enter state X *after* T_j has entered state Y. A strong dependency, denoted by (T_i, T_j, O, T_j.state = Y, X(T_i)), means that T_i *must* enter state X if T_j has entered state Y.

State dependencies are implemented by translating them into ECA rules. The run-time TMM supports ECA rules triggered by transaction events, i.e., occurrences of operations that change the state of transactions. For example, the execution of the Commit(T_1) operation causes the run-time TMM to create a commit event that may trigger an ECA rule. The TMM assembler identifies the type of each state dependency (backward, forward or strong) based on the syntax of the enabling and postconditions. Implementation of state dependencies is determined by their type. Let us assume that there is a commit dependency between transactions T_1 and T_2 such that transaction T_1 should commit only after transaction T_2 commits. This backward state dependency, expressed as $(T_1, T_2, O, T_1.state = Commit, Commit(T_2) < Commit(T_1))$, is implemented by first disabling operation Commit(T_1) then defining a rule that enables Commit(T_1) when the event Commit(T_2) occurs. If Commit(T_1) is issued before it is enabled, the run-time TMM traps Commit(T_1) and delays it until it becomes enabled. The disableOp(Commit(T_1)) is performed when T_1 is submitted for execution. Therefore, this backward state dependency is translated to

do disableOp(Commit(T_1))

if event Commit(T_2) occurs, then do enableOp(Commit(T_1))

Specifications of popular ETMs may be provided in the TSME as templates, and transaction model designers can combine components of different ETMs to form new ETMs. DOMS application programmers can write and submit extended transactions that behave according to an ETM supported by the TSME. If no such transaction model exists, the designers define a new ETM. The TSME also supports ETM *evolution* and *integration*, i.e., a TSME supported ETM can be extended to include new dependencies and we can integrate existing ETMs to produce more powerful transaction models, e.g., integrating nested transactions and sagas may result in a new transaction model such as the multi-transaction model.

Correctness dependencies include serializability, quasi-serializability, cooperative and temporal. To illustrate how they specify correctness dependencies, consider serializability (SR) dependencies. Specification of this dependency is based on transaction precedence relations defined in terms of direct and indirect conflicts. Let H be a history over a set of transactions $\tau = \{T_1, T_2, \cdots, T_k\}$, τ_c denote the set of committed transactions, and $T_i SR^*_{(\tau_c, O)} T_j$ denote that T_i SR-precedes (directly or indirectly) T_j in H. Therefore, in order to ensure serializability, every transaction must not directly or indirectly precede itself. This can be specified by the following dependency:

$\forall T_i (T_i, \tau_c - T_i, O, T_i.state = Commit, \neg[T_i SR^*_{(\tau_c, O)} T_i])$

In contrast to transaction state dependencies, ECA rules are not used for implementing correctness criteria. They argue that it is difficult to express the complex, transitive relationships between objects (inherent in correctness dependencies) using rules. Instead, they opt for using traditional scheduler technology for ensuring correctness criteria dependencies.

Comments

1. When dependencies are translated into ECA rules, the condition component is implicitly taken to be true. Therefore, it is not clear how the user can specify complex condition checking since he/she does not write the ECA rules.

2. They do not specify *how* events representing transaction states are detected, i.e., how the TMM actually traps the execution of operations such as begin and commit by transactions. Thus, although they have proposed using the active database paradigm as a mechanism for supporting transaction models, details such as how transactions are implemented, how transaction state changes are detected and how rules are enabled/disabled have not been addressed.

3. A non-uniform approach for enforcing dependencies is used, specifically, ECA rules are used for enforcing transaction state dependencies while traditional scheduler technology is used to enforce correctness dependencies.

4. Complex dependencies cannot be expressed due to the limited operators provided in the transaction specification language (the language used for specifying the enabling

and post conditions). Currently, only the conjunction $\wedge$, disjunction $\vee$ and the negation $\neg$ operators are provided. These are adequate for expressing simple transaction dependencies such as i) transaction T_1 cannot commit before transaction T_2 commits, ii) transaction T_1 cannot commit before transactions T_2 *and* T_3 commit, and ii) transaction T_1 cannot commit before either transaction T_2 *or* T_3 commit. These are respectively expressed by

- (T_1, {T_2}, O, T_2.state = Commit, Commit(T_2) < Commit(T_1))

- (T_1, {T_2, T_3}, O, T_2.state = Commit $\wedge$ T_3.state = Commit, (Commit(T_2) < Commit(T_1)) $\wedge$ (Commit(T_3) < Commit(T_1)))

- (T_1, {T_2, T_3}, O, T_2.state = Commit $\vee$ T_3.state = Commit, (Commit(T_2) < Commit(T_1)) $\vee$ (Commit(T_3) < Commit(T_1)))

However, consider process control environments where a dependency might state that transaction T_1 cannot begin execution until three hours after transaction T_2 commits. To model this, it is necessary to trap the event T_2 committing as well as provide some operator for monitoring time. Furthermore, some dependencies may be tedious to express given the limited number of operators provided. For example, consider a dependency stating that transaction T_1 cannot begin before *any two* of transactions T_2, T_3, T_4 and T_5 commit. Using the disjunction $\vee$ operator provided, this dependency can be expressed as

$(T_1, \{T_2, T_3, T_4, T_5\}, O, (T_2.state = Commit \wedge T_3.state = Commit) \vee (T_2.state = Commit \wedge T_4.state = Commit) \vee (T_2.state = Commit \wedge T_5.state = Commit) \vee (T_3.state = Commit \wedge T_4.state = Commit) \vee (T_3.state = Commit \wedge T_5.state = Commit) \vee (T_4.state = Commit \wedge T_5.state = Commit), ((Commit(T_2) < Commit(T_1)) \wedge (Commit(T_3) < Commit(T_1))) \vee ((Commit(T_2) < Commit(T_1)) \wedge (Commit(T_4) < Commit(T_1))) \vee ((Commit(T_2) < Commit(T_1)) \wedge (Commit(T_5) < Commit(T_1))) \vee ((Commit(T_3) < Commit(T_1)) \wedge (Commit(T_4) < Commit(T_1))) \vee ((Commit(T_3) < Commit(T_1)) \wedge (Commit(T_5) < Commit(T_1))) \vee ((Commit(T_4) < Commit(T_1)) \wedge (Commit(T_5) < Commit(T_1))))$

5. The fact that operations have a duration (i.e., are not instantaneous) is not taken into account. Each transaction operation like begin, commit etc. consists of a set of statements and it is not clear *when* (with respect to these statements) does an event take place. To clarify, consider the backward state dependency stating that transaction T_1 cannot begin execution before T_2 has begun execution. This will be expressed by $(T_1, T_2, O, T2.state = Begin, Begin(T2) < Begin(T1))$. The implementation of this dependency first issues disableOp(Begin(T_1)) and then when operation Begin(T_2) is issued it will be enabled again. However, it is not clear when Begin(T_1) is enabled with respect to the statements constituting Begin(T_2); is it enabled immediately after T_2 issues Begin *but before* execution of the constituent statements or is it enabled *after* execution of the constituent statements? This is an important issue in various transaction models such as the nested transaction model where parent and child concurrency is allowed. Here, in order to maximize concurrency, the child transaction should be enabled *immediately after* the parent transaction invokes the operation Begin *but before* execution of any of its constituent statements. For other transaction models, it is necessary to enable an operation after execution of all the constituent statements.

6. A *prevention* mechanism is utilized for enforcing dependencies.

7. Only three types of dependencies are supported : backward, forward and strong.

CHAPTER 5
RULE-BASED APPROACH TO SUPPORTING EXTENDED TRANSACTIONS

A transaction performs a number of operations during the course of its execution–some specified by the user and some performed by the system to guarantee certain properties. It is important to observe that the *semantics* of these operations differ from one transaction model to the other. For instance, the semantics of the *commit* operation in the traditional transaction model entails updating the log, making all updates permanent in the database, and releasing all locks held. This is in contrast to the commit of a subtransaction (in the nested transaction model), where all locks are inherited by the parent and the updates *not* made permanent until all superior transactions commit. As another example, consider the *lock-acquisition* operation in the traditional transaction model versus the nested transaction model. In the traditional model, a transaction can acquire an *exclusive-lock* on an object *only if* no other transaction holds *any* lock on that object. This is different from the nested transaction model where a subtransaction may acquire an *exclusive-lock* on an object even if one of its ancestor transactions holds a lock on that object. Moreover, some transactions perform operations which are very specific to that transaction model and not shared by other transaction models. As an example, the Split transaction model provides the *split* operation which when invoked causes the instantiation of a new top-level transaction and the delegation of some uncommitted operations to it. This *split* operation is not supported in other transaction models.

It is apparent that in order to support different transaction models in the *same* DBMS, one should not hardwire the semantics of operations such as commit, abort, read and write precisely because the semantics of these operations differ from one transaction model to the other. An obvious solution in an object-oriented environment is to create a transaction hierarchy and overload these operations or methods. Although this accomplishes the task of supporting different transaction models, it has a primary drawback, namely, it does not provide a general solution. In particular, the solution is specific to the object-oriented model rather than the system and thus cannot be readily applied to other DBMSs (e.g., relational). Another drawback of this approach is the performance penalty incurred as a result of resolving which method or operation needs to be executed[1]. Rather what is required is a flexible solution where it is possible to associate computations with operations (such as commit, abort, read and write) where these computations determine the semantics of the operation itself. Furthermore, for this mechanism to be effective and extensible, it should be independent of the programming model and the environment. And this is precisely what active capability *supported at the system level* offers.

Briefly, active capability allows for the *trapping* of particular operations, evaluation of a condition when the operation is trapped, and execution of an action if the condition evaluates to true. Therefore, by providing active behavior at the system level, it is possible to trap operations such as *acquire-lock* and associate with this operation a set of condition-action pairs, where each condition-action pair implements the semantics of this operation in a particular transaction model. For example, assume the traditional and nested transaction models are to be supported, then two condition-action pairs C_1A_1 and C_2A_2 will be associated with the operation *acquire-exclusive-lock*, where the former condition-action pair implements the semantics of this operation in the traditional transaction model and the latter condition-action pair implements the semantics of this operation in the nested transaction model. C_1A_1 checks to see that no other transaction holds any lock on the object in question and if this evaluates to true it grants the lock to the requesting transaction, otherwise it blocks the transaction on a semaphore. On the other hand, C_2A_2 checks to see if all the transactions which hold any lock on the object in question are superiors of the requesting transaction, and if this evaluates to true the lock is granted, otherwise

[1]Notice that no room for optimization exists at this level since this is a language dependent issue.

the transaction is blocked on a semaphore. The association of condition-action pairs with operations is depicted in Figure 5.1.

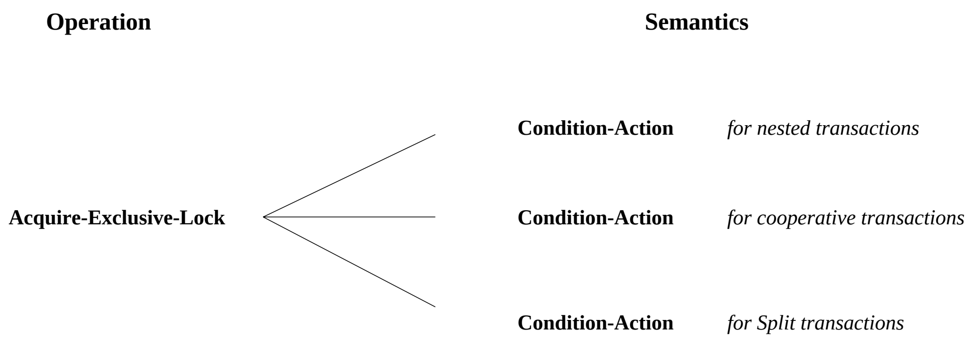


Figure 5.1. Condition Action Pairs

This chapter is organized as follows. Section 5.1 begins by defining the notion of active behavior in the context of DBMSs. Section 5.2 then follows by presenting our approach to realizing an extensible transaction management facility using the active database paradigm; specifically, we show how it is possible to utilize the active database paradigm at the *system level* to achieve the semantics of various transaction models in the same underlying DBMS.

5.1 Active Databases

During the past years DBMSs have undergone dramatic changes as a result of the increasing requirements of newer applications. Conventional record-oriented database systems are subject to the limitations of a finite set of data types and the need to normalize data. These limitations have led to the evolution of a new paradigm, namely object-oriented databases (OODBMS), which offer increased modeling power, flexible abstract data-typing facilities and the ability to encapsulate data and operations via the message metaphor. Despite the ability to model complex objects and relationships, these OODBMSs lack some of the requirements of a large class of new applications, specifically those requiring monitoring of situations and responding to them automatically, possibly subject to timing constraints.

Active databases have been proposed to meet some of the requirements of non-traditional applications. Active OODBMSs extend the normal functionality of OODBMSs with support for monitoring user-defined situations and reacting to them without user or application intervention. These DBMSs continuously monitor situations to initiate appropriate actions in response to database updates, occurrence of particular states or transition between states, possibly within a response time window. The emergence of this trend of active OODBMSs serves a large variety of applications such as computer integrated manufacturing (CIM), process control, battle management, and network management. Furthermore, active databases provide an elegant means for supporting integrity constraints, access control, maintenance of derived data, and materialized views and snapshots.

Active behavior has been used to connote different behavior in various contexts in diverse areas of computer science. Morgenstern [34] used the term *active database*, perhaps for the first time, to describe a system that supports automatic update of views, and derived data as base data are updated. In AI community the term *active object* is used either for active knowledge representation and inference mechanism or for achieving intelligent behavior and concurrent computation. The programming language community uses the term *active object* in order to structure concurrent applications in object-oriented programming languages. Ishikawa [28] used the term *active object* to distinguish real-time objects from others which have timing constraints. Osterbye [38] used the term *active object* to represent objects with two types of demons which are executed when the object is accessed. It resulted from the generalization of access-oriented programming.

The key distinction between an active and a passive object, as conveyed in the database literature, lies in an active object's ability to initiate asynchronous actions, as a separate thread of execution, without necessarily receiving messages. Typically, "passive" objects respond to messages through a *synchronous* interface; they *receive a message* and based on its interpretation *then perform* some operations and return a result.

In summary, the term *active* has been used to convey concurrency, asynchronous behavior, and parallelism of active objects, intelligent behavior of agents/actors, or active capability of a system. In other literature similar notions are elaborated without using the term *active* explicitly. Rules, also referred to as triggers and alerters, have been proposed to provide active functionality in OODBMSs. Rules, in the context of an active DBMS, consist primarily of three components: an event, a condition, and an action. An event is an indicator of a happening (either simple or complex). Events are recognized by the system or signalled by the user. For example, database events such as insert, delete, and update are detected by the OODBMS. The condition specifies an optional predicate over the database state which is evaluated when its corresponding event occurs. The conditions to be monitored may be arbitrarily complex and may be defined not only on single data values or individual database states, but also on sets of data objects, transitions between states of materialized/derived objects, trends and historical data. Actions are the operations to be performed when an event occurs and its associated condition evaluates to true. Actions may be programs which may in turn cause the occurrence of other events. Once rules are specified declaratively to the system, it is the system's responsibility to monitor the situations (event-condition pairs) and execute the corresponding action when the condition is satisfied without any user or application intervention. Figure 5.2 depicts how an application interacts with an active DBMS. Basically, an application still interacts with the database by executing operations which are ultimately translated into read and write operations against the database. In addition, the application can define a set of rules which are given to an activity management module which is responsible for keeping a *watchful eye* on the database in order to detect the occurrence of events, and then to evaluate the conditions and possibly execute the actions. The advantage of using rules as a means of providing active behavior is the freedom from explicitly hard-wiring code which checks the situations being monitored into each program that updates the database.

Figure 5.2. Active DBMS Architecture.

5.2 Our Approach

Our approach for supporting extended transaction models is based on the observation that the added functionality provided by the active database paradigm (in the form of event-based or ECA rules) cannot only be used by applications to achieve application level functionality such as constraint management, but also for supporting system functionality. Up to this point, most of the efforts on active database support have considered usage of ECA rules for user-defined event-condition-action rules that can be specified to augment application code (e.g., for expressing integrity constraints). As the active database technology is maturing (as evidenced by a number of research prototypes), there is clearer understanding of the implementation techniques, data structures required, and optimization techniques. This knowledge is essential for using this capability at the systems level.

We propose to use active databases as a mechanism for specifying and enforcing the behavior of different transaction models. In particular, we provide the user with the ability to *construct* the required transaction semantics on an application by application basis; specifically, we show how each transaction model can be translated into a set of ECA rules which can be activated or deactivated by users. For example, consider the nested transaction model where the commit of a top-level transaction can occur *only after* the termination of all of its subtransactions. In this case, the event component of the rule corresponds to the detection of a request to commit by a top-level transaction. When this event is detected, the condition checks whether the children of the transaction requesting to commit are still active, i.e., have not yet committed or aborted. If the condition evaluates to true (i.e., at least one child has not yet terminated), the action will postpone the commit of the top-level transaction until its children have terminated thereby enforcing the behavior of the nested transaction model. For concreteness, we show how different transaction models can be specified in the context of Sentinel, an active OODBMS developed at UF. However, our approach is general enough to be applied by any active DBMS.

Active database paradigm can be used in a number of ways to support flexible transaction models. Below, we examine these alternatives and discuss the merits of each approach, ease of its implementation, and the extent to which it can support extended transaction models.

5.2.1 Alternatives for Supporting Extended Transaction Models

The alternatives for supporting different transaction models given an active DBMS can be broadly classified into the following approaches:

1. Provide a set of rules that the user can use from within applications to get the desired transaction semantics. This approach assumes that the underlying DBMS supports some transaction model. In this approach the rules are defined on the operations provided by the built-in transaction model, e.g., commit, abort etc. Consequently, the semantics of *only* those transaction models which are very *similar* to the the built-in transaction model can be expressed in this approach. The desired transaction semantics is obtained by the user by enabling any one of the rule sets provided. For example, we assume that there is a set of rules for nested transactions that can be enabled by a command giving the user the semantics of nested transactions. Minimal user commands such as begin- and end-subtransaction are assumed. Similarly, another set of rules will provide the semantics of Sagas. Without any loss of generality we shall assume that rules are in the form of ECA rules, i.e., event, condition and action rules (along with coupling modes, event contexts, priority etc).

   One advantage of this approach is that new rule sets can be defined (of course by a DBA or a DBC) and added to the system. It may also be possible for the educated user to add additional rules to slightly tweak the semantics of a transaction model. A limitation of this approach is that the set of rules defined are over the events of the conventional transaction model supported by the system, e.g., commit, abort, etc., and thus limits the number of transaction models which can be expressed using these events.

2. Identify a set of critical events on the underlying data structures used by a DBMS (such as the operations on the lock table, the log, and deadlock and conflict resolution primitives) and write rules on these events. This approach does not assume any underlying transaction model. This approach can be used to support different transaction models including the traditional transaction model. In this approach, system level ECA rules are defined on data structure interfaces to support flexible transactions.

   A distinct advantage of this approach is that it will be possible to support workflow and newer transaction models irrespective of whether they are extensions of the traditional transaction model. To elaborate, the rules are now defined on low-level events which act on the data structures directly thereby providing finer control for defining transaction semantics. For instance, a rule can be defined on lock table events such as *acquire-lock* and *release-lock*. This is in contrast to defining rules on high-level events such as commit, abort etc. Another advantage is that a DBMS can be configured using a subset of the transaction models available at the system generation time. This approach may be able to offset the performance disadvantage currently observed in active database systems. The system designer will be in a better position (relatively) to support or extend transaction models[2].

   This approach is similar to the one taken in Wells et al. [46]. They introduce a flexible and adaptable tool kit approach for transaction management. This tool kit enables a database implementor or applications designer to assemble application-specific transaction types. Such transaction types can be constructed by selecting a meaningful

[2]We would like to point out that the use of ECA rules by themselves will not make the system completely flexible. However, we do believe the process of identifying primitive events, details of conditions/actions and writing these rules will make us reexamine the current architecture and the data structures to progress towards a modular systems architecture.

subset from a starter set of basic constituents. This starter set provides, among other things, basic components for concurrency control, recovery, and transaction processing control.

3. This is a generator approach using either the first or the second alternative. In this approach a high-level specification of a transaction model (either by the user or by the person who configures the system) is accepted and automatically translated into a set of rules. The specification is assumed at compile time so that either rules or optimized versions of code corresponding to the rules are generated. The advantage of this approach is that the burden of writing rules is no longer on users of the system.

We propose to adopt the second approach outlined above as a mechanism for specifying and enforcing extended transaction models. The rationale for adopting this alternative is that this approach offers *fine-grained control*. This control provides the necessary flexibility for specifying existing transaction models, newer transaction models as they become available and arbitrary transaction semantics thereby resulting in a highly extensible system. Before, we give details of the second alternative outlined above, it is first necessary to identify the advantages of using the active database paradigm as a mechanism for expressing and enforcing the semantics of various transaction models. This is important since many other alternatives may exist for solving this problem and thus these advantages helps in providing metrics for comparing different viable solutions. The advantages of using this paradigm for supporting different transaction models are summarized below:

- Provides a model and environment independent mechanism for supporting various transaction models.

- Provides a *uniform* mechanism for achieving *system extensibility*. The utility of active behavior at the systems level can also be applied to achieve additional functionality such as load balancing.

- It is a well understood and established paradigm and thus can be implemented, analyzed, and optimized with relative ease. The utility of this paradigm can be observed by the presence of this capability in almost all commercial models, its introduction into SQL3, and the number of research prototypes being developed.

- The availability of expressive event specification languages like Snoop, Samos, Ode, and Reach that allow sequence, conjunction and time related events can be beneficial for modeling interdependencies amongst transactions. For example, complex dependencies such as transaction T_1 cannot begin execution until any of transactions T_2, T_3 or T_4 commit, followed by the commit of transactions T_5 and T_6, can be expressed using the expressive event specification languages mentioned above. In addition, these languages can be beneficial for modeling some of the synchronization aspects of workflow applications.

Our approach for supporting a given transaction model T_x using active capability is essentially a three step process :

1. Identify the set of operations executed by transactions in the model under consideration. Both application visible and internal operations are taken into account. For example, application visible operations such as *begin transaction* and internal operations such as *acquire lock* are considered. Some of these operations are treated as *events*, i.e., their execution is *trapped* by the active DBMS. It should be emphasized that not all events detected are associated with operations implemented in the system. Rather, these events can be abstract or external events.

2. The second step involves identifying the condition which needs to be evaluated when an event occurs (e.g., checking for conflicts at lock request time) and the action to be performed if the condition evaluates to true (e.g., granting the lock to the requesting transaction). The events, conditions and actions yield pools of events, conditions,

and actions, respectively, which are stored in the DBMS. These pools, depicted in Figure 5.3, form the building blocks from which rules are constructed.

3. The final step involves combining an event, a condition and an action to compose an ECA rule. Each ECA rule defines the semantics of a smaller unit of the transaction model under consideration. For instance, an ECA rule may define the semantics of the *acquire lock* operation. This process is repeated until a *rule set* defining the entire semantics of a transaction model is built. We allow for the cascading of rule execution. This occurs when the *action* component of a rule raises event(s) which may trigger other rule(s). Cascading of rules is utilized for implementing nested transactions and is shown in chapter 7.

This approach allows sharing of the building blocks in several ways. Events, conditions, and actions are shared across rules sets composed for different transaction models. In addition, intermediate rules can also be shared by other rules. Although Figure 5.3 shows a single level for clarity, a hierarchy of rules is constructed from the building blocks. The overlap of events, conditions and actions for different rule sets clearly indicates the modularity and reusability aspect of our approach. This is further substantiated in the section on implementation details.

To summarize, our approach encapsulates the semantics of a transaction model into ECA rules. These rules are derived from the analysis of each transaction model as well as examination of their similarities and differences. This encapsulation is done at the level of significant operations (e.g., begin-transaction, commit) that can be treated as events and/or at the level of internal operations on data structures (e.g., lock-table). Once the semantics of a transaction model is composed in terms of these building blocks, rules are written for each block. The availability of begin and end events are useful to model the semantics without having to introduce additional events. Also, the availability of coupling modes and composite events are used to avoid explicit coding of control as much as possible.

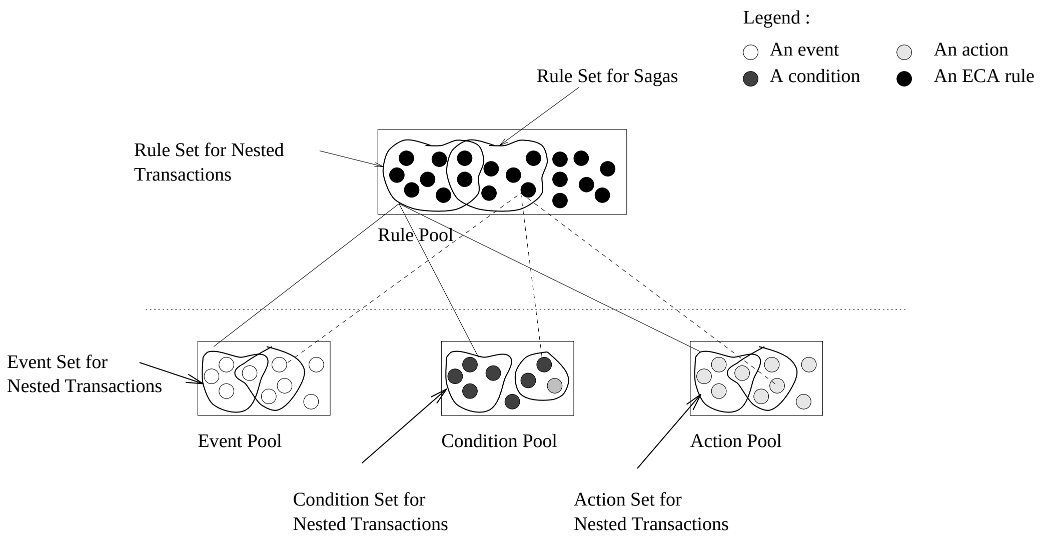

Legend :
An event
An action
A condition
An ECA rule
Rule Set for Sagas
Rule Set for Nested Transactions
Rule Pool
Event Set for Nested Transactions
Event Pool
Condition Pool
Action Pool
Condition Set for Nested Transactions
Action Set for Nested Transactions

Figure 5.3. Rule Composition

The mechanism described above can also be applied for customizing auxiliary behavior. By trapping the operations that are executed by applications, it is possible to perform *auxiliary* actions as required by the user/system designer (i.e., other than those defining the transaction semantics). A good example of this is in systems where optimal performance is achieved when the number of transactions in the system does not exceed a particular threshold (e.g., load balancing and buffer sizes). Therefore, it is necessary to check the number of transactions and not allow the threshold to be exceeded. This can be accomplished by trapping the operation *begin transaction* and checking the number of active transactions at that point. If the number is found to be less than the threshold, then allow the transaction to continue execution, otherwise either abort the transaction or make it wait. Similarly, in banking applications there may be a limit on the number or amount of withdrawals in a day. By defining a rule which is triggered upon detection of the *begin* operation, it is possible to check the number or amount of withdrawals appropriately and either continue or abort the transaction. To summarize, not only does the active database paradigm allow for the specification of transaction semantics but arbitrary semantics as well in an extensible manner.

5.2.2 Benefits of Our Approach

Our approach, which is essentially use of the active database paradigm at the system level and the rule composition process, provides the following benefits:

- Applications can avail the semantics of a particular transaction model T_x by *enabling* the rule set defining the semantics of T_x. Disabling a rule set will eliminate runtime overhead associated with the firing of rules in that rule set.

- Using the proposed approach, different applications can realize desired transaction semantics by *choosing* the appropriate rule set. It is also possible to make individual rules available to applications by the DBC in an appropriate manner. This usage requires a good understanding of the internals and hence needs to be used cautiously.

- More importantly, realizing the semantics of a particular transaction model entails defining a rule set to be used by transactions adhering to this model. As *reusability* of rule sets among different transaction models is a key aspect of this approach, it may be possible to define a new rule set using existing rules.

- Rule sets to support various transaction models can be provided by the DBC. The number of transaction models supported can be controlled by the number of rule sets currently available to applications. Application specific auxiliary semantics can be provided by the DBC by writing rules specific to an application class/environment.

- It is possible to *configure* a DBMS with one or more rule sets and optimize the rule sets for efficiency. In other words, efficiency need not necessarily be sacrificed if only one or a small set of transaction models are desired. This can be achieved by using the subscribe/unsubscribe functionality, or by compiling desired rules at configuration time [31]. Subscribe and unsubscribe allows us to decouple rules (condition-action pairs) from events, thereby reducing runtime overhead.

- When a new event, such as delegate, is introduced for modeling the semantics of new transaction models, multiple rules can be associated with that event to provide different semantics. These set of rules become part of the pool of rules available for grouping.

5.2.3 Contributions

Before presenting the details of our research results, we first list the contributions of our work:

- We propose the use of ECA rules for obtaining the semantics of transaction models. Rule sets on significant events specify a transaction model. Both events and rules can be added without changing the semantics of currently supported transaction models.

- We allow the user to define ECA rules on the *underlying data structures* in the DBMS (such as the transaction table, object table and log) thereby providing fine-grained control to the user. This provides a powerful facility since the user can *build* or equivalently *construct* arbitrary transaction semantics rather than being restricted to those transaction models which can be expressed using high-level transaction primitives and dependencies.

- We provide a framework which separates the definition of rules (which define the transaction semantics) from the application code itself. This permits applications and transaction semantics to be modified independently of each other as well as use of existing applications without major modifications.

- Rules defining transaction semantics can be activated and deactivated *dynamically* thereby enabling the transaction semantics to be modified dynamically. This is an important requirement for testing the applicability of different transaction models for the same application.

- We show how various transaction models can be translated into a set of ECA rules in the context of Sentinel, an active OODBMS developed at UF. However, our approach is general enough to be applied to any active DBMS (relational or otherwise).

CHAPTER 6
DESIGN DETAILS

Sentinel, an active OODBMS developed at UF was used as the underlying DBMS for implementing the traditional transaction model, nested transactions and Sagas using ECA rules. Sentinel was developed by incorporating active capability into Zeitgeist, an OODBMS developed at Texas Instruments. In the following sections, we first begin by describing Zeitgeist with special emphasis given to its transaction manager (TM) architecture. We then proceed by explaining how active behavior was incorporated into Zeitgeist thereby producing Sentinel. A discussion of the adequacy of Sentinel for supporting various transaction models is then given followed by examination of the alternatives for supporting extended transaction models in an *object-oriented* environment.

6.1 Zeitgeist

The OODBMS Zeitgeist consists of the following components: i) Persistent Object Store (POS), ii) Object Management System (OMS), ii) Primitive Object Query Language (OQL) and iv) Flat or traditional Transaction Manager (TM). Figure 6.1 depicts the Zeitgeist modules. Zeitgeist uses the C++ language to define the OO conceptual schema as well as to manipulate the internal object structure. Below, we describe the components pertinent to our implementation, namely, the POS, OMS and TM components.

- **Persistent object store.** Zeitgeist provides two methods for achieving persistence, namely, using a file-based structure on the Unix platform or using an underlying storage manager for persistence. The file-based version of Zeitgeist neither supports concurrency control nor recovery. Therefore, all transactions must be executed sequentially and no provision for recovery when faults occur is given. The other method

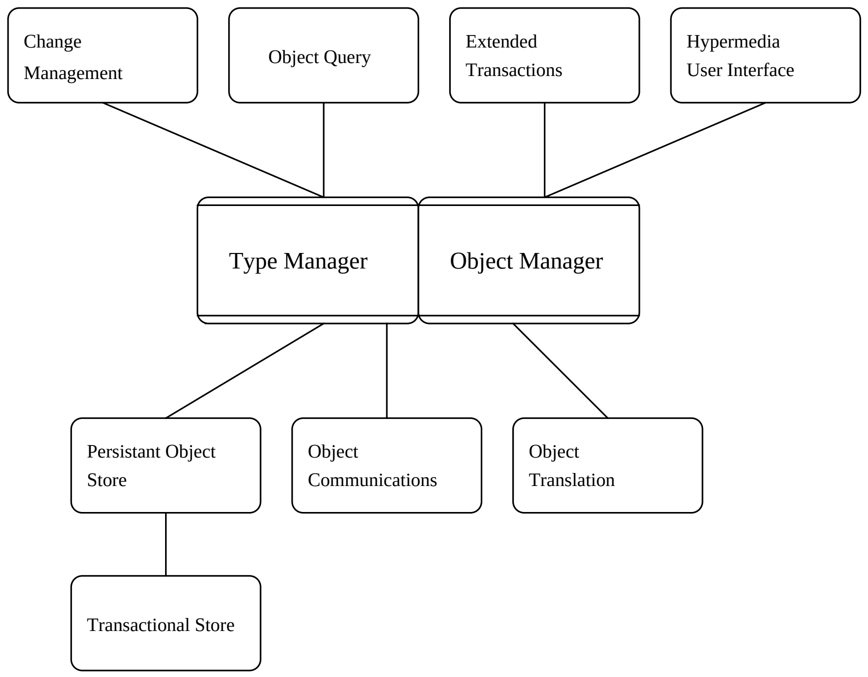


Figure 6.1. Zeitgeist Modules.

for achieving persistence is by translating C++ objects into record format and storing them in an underlying storage manager. Although, Zeitgeist originally only used Oracle as an underlying storage manager, interfaces for using both Ingres and Sybase were developed at UF. In our implementation we used Ingres as the underlying storage manager. However, this is not a limitation since it can be easily ported to other storage managers, e.g., Oracle. Concurrency is provided by maintaining a *multi-level* transaction manager where the bottom-level transaction manager belongs to the underlying storage manager and the top-level transaction manager is maintained in a shared memory segment at the client level. The top-level transaction manager provides concurrency control only while recovery is provided by the lower-level transaction manager, namely, Ingres's transaction manager.

Object persistence is achieved by using three relations in Ingres viz. **Groups, Value** and **Refto**. These three tables are created and initialized by executing scripts at the unix command line. These scripts connect to Ingres and execute SQL statements.

Note that these tables can be directly accessed for debugging purposes. The *Groups* table controls the allocation of object numbers as well as contains the information for each storage group and the time when it was last modified. This relation consists of the attributes SGNO, NEXTNO, LOC, and USER_COMMENT. This table has a unique ascending index on SGNO. The next table, *Value*, holds object values, i.e., this is where the translated C++ form of the object is stored. The *Value* table consists of the attributes SGNO, OBNO, TIME, SEQNO, OBJSIZE, ASGNO, AOBNO, ATIME, DSGNO, DOBNO, DTIME, SACNT, UACNT, XCNT, and VALUE. The last table, *Refto*, holds externally referenced OIDs, i.e., it maintains the external references of an object. The attributes of this table are SGNO, OBNO, TIME, SEQNO, XSGNO, and XOBNO.

- **Object manager.** The object manager is responsible for acquiring locks on an object as well as fetching an object from the POS and installing it in process memory. The object manager maintains a data structure referred to as the encapsulated object (EO). This data structure keeps track of the lock mode in which each object is held as well as which transaction holds the lock. When a transaction requests an object for the first time, the object manager fetches the object from the POS and installs it in main memory. All subsequent requests for that object (by the same transaction) result in the object manager first checking for the presence of the object in the EO data structure followed by it deciding whether the request should be granted or the lock upgraded etc.

- **Transaction manager.** Zeitgeist's TM is implemented using three main classes, namely, the *zgt_ht* class, the *zgt_tx* class, and the *zeitgeist* class. Figures 6.3, 6.4 and 6.5 respectively show the class definitions of each of the above mentioned classes. The entire TM's data structures are maintained in a shared memory segment thereby allowing it to be accessed by multiple clients. Access to the TM's data structures by

```
static struct zgt_hlink
{
        char lockmode; // the lockmode in which the object is held
        long sgno; // the storage group number in which the object is stored
        long obno; // the object identification number of the object held
        long tid; // the transaction identifier number of the transaction holding the object
        int pid; // the process identification number of the transaction holding the object

        zgt_hlink * next; // links nodes hashed to the same bucket
        zgt_hlink *nextp;  // links nodes of the same transaction, i.e., all locks held by transaction

}
```

Figure 6.2. The structure showing the locks held by a transaction.

clients is regulated using a specific semaphore. First, the user executes the program *zgt_init* which is responsible for creating and attaching the shared memory segment, for allocating and initializing a specific number of semaphores, and for creating and initializing the TM's data structures. The semaphores are used to regulate access to the TM's data structures (since this is a critical section) as well as for transactions to block on when their requests cannot be granted. The *zgt_init* program takes four parameters at the command line: the user identification number, the number of semaphores, the size of the shared memory segment, and the number of seconds denoting the time interval between checking for deadlocks. With respect to the number of semaphores, the lower bound is 3 and the upper bound is 64. If the user specifies the number of semaphores to lie outside the latter range, then the default number of 16 semaphores is allocated. The lower bound on the shared segment size is 65536 bytes while the upper bound is 1048576 bytes. If the user specifies a shared memory segment size which does not lie within this range, then the default size of 655360 bytes is used.

Figure 6.3 contains the class definition for the *zgt_ht* class. The data members are *sm, lastid, lastr, head, size* and *mask*. The first data member, *sm*, points to the start of the shared memory segment. *Lastid* is the value of the last transaction identifier which

```
class zgt_ht

{
    public:

        friend class zgt_tx;
        friend class wait_for;

        // state

        char *sm;  // pointer to the start of the shared memory segment
        long lastid;  // value of last transaction identifier assigned
        char *lastr; // pointer to the first transaction object in the hash table
        zgt_hlink ** head; // origin of has table
        int size; // length of hash table array
        int mask; // bit mask used in hash function


        // methods

        zgt_hlink * find (long, long);
        zgt_hlink * findt (long, long, long);
        int add (zgt_tx *, long, long, char);
        int remove (zgt_tx *, long, long);  // remove a lock entry
        int hashing (long sgno, long obno) {return ((++sgno)*obno)&mask;};
        // constructors and destructors

        zgt_ht (int ht_size = ZGT_DEFAULT_HASH_TABLE_SIZE, zgt_shmem *shm = NULL);
        ~zgt_ht();
}
```

Figure 6.3. The Hash Table Class.

was assigned to a transaction. This data member ensures that each transaction is assigned a unique transaction identifier. *Lastid* is simply incremented to assign a new transaction identifier. The *lastr* data member points to a chained list of transaction objects where the structure of each transaction object is given in Figure 6.4. The *head* data member points to the start of a lock hash table. The lock hash table consists of a number of buckets where each bucket maintains a linked list of locks held by transactions. To elaborate, each item in the linked list contains information regarding which object is held by which transaction as well as the lock mode in which it is held. The *size* data member contains the value of the size of the lock hash table, specifically the number of buckets assigned. Finally, the *mask* data member contains a bit value used to determine where an object stored in a particular storage group should hash to in the lock hash table.

```
class zgt_tx

{

    public:

        friend class zgt_ht;
        friend class wait_for;

        long tid;    // the transaction identifier
        int pid;     // the process identification number of the transaction
        long sgo;    // the storage group number where the object on which the transaction is blocked is stored
        long obno; // the object number of the object on which the transaction is blocked
        char status;  // the current status of the transaction
        char lockmode; // the lockmode requested for the object on which the transaction is blocked
        int semno; // the semaphore number on which the transaction is queued
        zgt_hlink * head;  // this points to a linked list of the locks currently held by the transaction
        zgt_tx * nextr; // this points to the next transaction hashed to this same bucket


        // methods

        zgt_hlink *others_lock(zgt_hlink *, lonh, long);
        int free_locks();
        int remove_tx(zgt_shmem *);
        long get_tid() {return tid;}
        long set_tid(long t){tid = t; return tid;}
        char get_status() {return status;}
        int set_lock(long, long, char);
        int set_lock_no_wait(long,long, char);
        int upgrade_lock_no_wait(long, long, char);
        int end_tx();
        int cleanup();
        zgt_tx(zgt_shmem *);
        ~zgt_tx(){};
}
```

Figure 6.4. The Transaction Class.

With respect to the methods of the *zgt_ht* class the methods are *find, findt, add* and *remove*. The *find* method takes two arguments as its parameters, namely, a storage group number and an object number. This method first hashes the storage group number and the object number to determine which bucket in the lock hash table this object should reside. It then performs a linear sequential search in that bucket to find the *first* occurrence of that object and storage group number in the linked list. If one is found, a pointer to that *zgt_hlink* element is returned, otherwise a null pointer is returned. This method basically determines whether *any* transaction holds *any* lock on that particular object. The next method, *findt*, takes three arguments, specifically, a transaction identifier, a storage group number and an object identifier. This method first hashes on the storage group number and the object number to determine the

```
class zeitgeist
{
    public :

    // methods

    int abort_transaction();
    int begin_transaction();
    int commit_transaction();
    int shutdown();
    int startup();

    .
    .
    .
}
```

Figure 6.5. The Zeitgeist Class.

bucket in the lock hash table. It then performs a sequential search in the bucket to determine whether the specified transaction (given as an argument to this method) holds *any* lock on the given object and storage group. If a zgt_hlink is found, then a pointer to it is returned, otherwise a null pointer is returned. The next method *add* takes three arguments, namely, a pointer to a transaction object, a storage group number, an object number and a lock mode. This method first acquires a piece of memory from the shared memory segment. The structure of this piece of memory is a zgt_hlink object. It first hashes on the storage group number and the object identifier to determine which bucket the object is to reside in and then adds it at the top of the linked list of zgt_hlink objects in the bucket. Subsequently, it updates the linked list of locks held by the transaction object to reflect the acquisition of this lock. To summarize, this method updates the TM's data structures to reflect the acquisition of a lock by a transaction. The last method, *remove*, takes three parameters, namely, a transaction object, a storage group number and an object identifier. This method basically removes a zgt_hlink from the linked list of locks held by a transaction. Therefore, this method essentially performs the lock release operation by a transaction. Note that all methods which update the TM's data structures must first acquire an exclusive semaphore so as to avoid simultaneous access to these shared data structures.

The *zgt_tx* class is used to represent all transactions in the system. The class definition is given in Figure 6.4. The data members of this class are *tid, pid, sgno, obno, status, lockmode, semno, head,* and *nextr*. The *tid* and *pid* respectively denote the transaction identifier and the process identifier of the transaction. The *sgno* and *obno* denote the storage group number and the object identifier for which the transaction is currently waiting. This implies that the transaction is currently blocked. The *status* data member maintains the transaction state information which can be any one of the following four states:

1. **Active.** This implies that the transaction is currently executing. This state is denoted by the character **P**.
2. **Wait.** This state implies that the transaction has been initiated and has actually started execution but is currently in a wait state. This state arises when a transaction requests a particular lock on an object and this lock cannot be granted due to a conflict. In this state, the transaction is blocked on a semaphore. This state is denoted by the character **W**.
3. **Abort.** This state implies that the transaction has aborted. This arises when a failure occurs or due to deadlock resolution. In this state, all the operations performed by the transaction are rolled back. This state is denoted by the character **A**.
4. **End.** This state denotes that the transaction has successfully completed all of its operations. This state is denoted by the character **E**.

The *head* data member points to a linked list of *zgt_hlink* objects. Each zgt_hlink object maintains information about a lock held on an particular object. Therefore, the *head* data member points to a linear list of all the locks held by a transaction. The last data member, *nextr*, points to the next transaction object. Therefore, all transaction objects are maintained as a linear list.

The methods of this class are *others_lock, free_locks, remove_tx, get_tid, set_tid, set_lock, get_status, set_lock_no_wait, upgrade_lock_no_wait, end_tx, cleanup, zgt_tx* and *~zgt_tx*. The first method, *others_lock*, takes three parameters, basically a pointer to a bucket in the lock hash table, a storage group number and an object identifier. This method determines whether *any* other transaction holds *any* lock on the object in question. The method *free_locks* frees all locks held by a transaction. While traversing sequentially through the locks held by the transaction, any transaction which is blocked because it needs the just released object is unblocked and allowed to proceed. The method *remove_tx* first checks that no locks are held by a transaction and if this is found to be true, the transaction object is removed from the linked list of transactions and the memory it utilized is returned to the free list of shared memory available. The methods *get_tid, set_tid* and *get_status* return the transaction identifier, set the transaction identifier, and return the status of the transaction, respectively. The methods *set_lock* and *set_lock_no_wait* both take three parameters, namely, the storage group number, an object identifier and a lock mode. The former method attempts to acquire the specified lock on the given object and if it is unable to do so, it returns an error. The latter method behaves similarly except if it is unable to acquire the lock on the object, the transaction is blocked on a semaphore and its status is changed to the wait state. The method *upgrade_lock_no_wait* takes three parameters, namely, a storage group number, an object identifier and a lock mode. This method attempts to upgrade the lock held by a transaction and if it is unable to do so, it returns an error message. The method *end_tx* first frees all locks held by a transaction, removes the transaction object from the linked list of transactions and returns all released memory to the free list of shared memory available. The *cleanup* method kills all transactions in the linked list while releasing and returning all memory to the free list of shared memory available. The constructor *zgt_tx* takes as a parameter a pointer to the shared memory segment and acquires a piece of memory for creating and initializing a new transaction object. Finally, ~*zgt_tx* is the class destructor. Notice that this class only implements the operations associated with locks, i.e., lock acquisition, lock release, lock upgrade etc. Other operations associated with a transaction like begin, commit and abort are not implemented here but rather implemented in the Zeitgeist class.

The structure zgt_hlink is depicted in Figure 6.2 and is used for representing the locks held by a transaction. This structure consists of the data members *lockmode, sgno, obno, tid* and *pid.* The *lockmode* is a character denoting the mode in which the object is held. Zeitgeist supports three locks modes:

– **Read only (RO).** This lock mode is not upgradable. A transaction is not required to acquire a lock on an object if the lock mode requested is read only.

– **Shared (S).** This lock mode implies that the transaction wants to read the object. More than one transaction can have a shared lock on the same object at the same point in time. This lock is upgradable. This lock can be upgraded to an exclusive lock provided no other transaction holds a shared or exclusive lock on that object.

– **Exclusive (X).** This lock mode implies that a transaction has both read and write access on an object. Only one transaction may hold an object in exclusive mode at a time.

The Zeitgeist class is partially shown in Figure 6.5. The methods of this class pertinent to transaction processing are *abort_transaction, begin_transaction* and *commit_transaction.* The first method, *abort_transaction*, rolls back all the updates made by a transaction. This takes place when a failure occurs. The *begin_transaction* method performs all initialization procedures so as a transaction may subsequently begin execution. Finally, the *commit_transaction* method makes all updates performed by a transaction permanent in the database.

The relationship between the above mentioned three classes is illustrated in Figure 6.6.

6.2 Overview of Sentinel

Sentinel [11, 12, 16, 17, 13, 44, 2, 4, 14, 15, 29, 45] is an active object-oriented DBMS that seamlessly integrates ECA rules into the object-oriented paradigm. The Sentinel architecture is an extension of the *passive* Zeitgeist architecture [37]. The Zeitgeist class hierarchy was augmented with new class definitions which are necessary for supporting active capability. Figure 6.7 depicts the class hierarchy of Sentinel; specifically, the classes introduced are the *Reactive*, *Notifiable*, *Event*, *Rule* and *Event Detector* classes.

In Sentinel, objects are classified into three categories: passive, reactive and notifiable. **Passive objects.** These are conventional objects which receive messages, perform some operations and then return results. They do not generate events. An object that needs to be monitored (by informing other objects of its state changes) cannot be passive. **Reactive objects**, on the other hand, are objects that need to be monitored (i.e., on which rules will be defined). A reactive object can declare any, possibly all, of its methods as an *event generator*. All methods declared as event generators constitute a reactive object's *event interface*. Once a method is declared as an event generator, its invocation will generate a primitive event. The primitive event can be generated either *before* or *after* the execution of the method depending on which *event modifier* was specified by the user. The event will be generated before execution and after execution if the user specifies the *begin* and *end* modifier, respectively. In addition, if the user specifies both modifiers then two primitive events will be generated, one before execution and one after execution of the respective method. Lastly, **Notifiable objects** are those objects that are capable of being informed

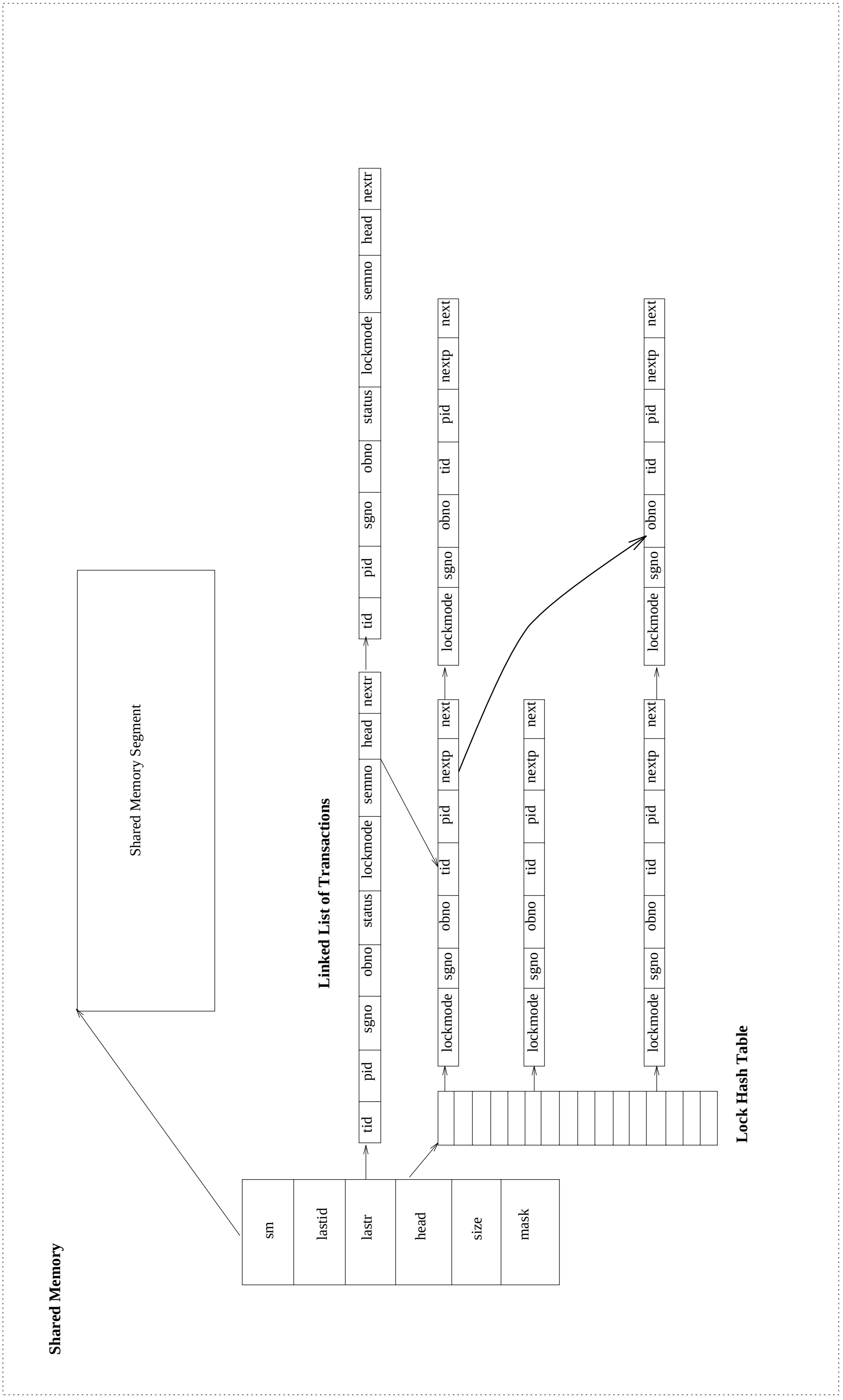


Figure 6.6. Architecture of Zeitgeist's Transaction Manager.

of the events produced by reactive objects. Therefore, notifiable objects become aware of a reactive object's state changes and take appropriate measures (by evaluating conditions and executing actions) in response to those state changes. Notifiable objects *subscribe* to the primitive events generated by reactive objects. After subscription, the reactive objects propagate their generated primitive events to the notifiable objects. Events and rules are examples of notifiable objects. Rules receive events from reactive objects, send them to their local event detector, and take appropriate actions. Event detectors receive events from reactive objects, store them along with their parameters, and use them to detect primitive and complex events. In the following paragraphs we briefly outline the implementation of the *Reactive*, *Notifiable*, *Event* and *Rule* classes. The reader is referred to Anwar et al. [2] for a detailed implementation of these classes.

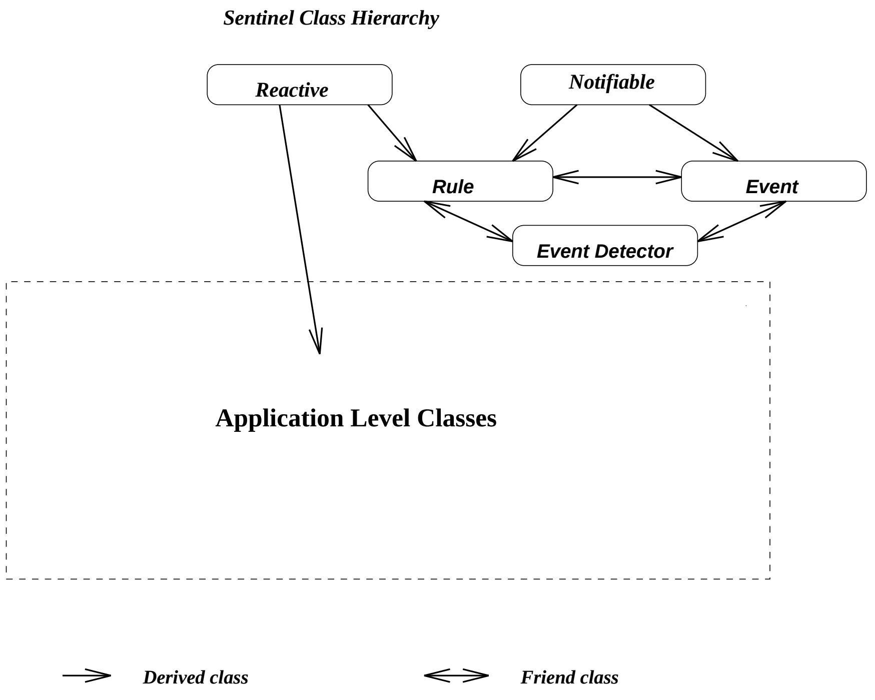


Figure 6.7. Sentinel class hierarchy

Reactive class. The public interface of the Reactive class consists of methods by which objects acquire reactive capabilities. For an object to be reactive, i.e., have the ability to generate primitive events when methods in its event interface are invoked, it must be an instance of a class derived from the Reactive class[1]. Subclasses of the Reactive class will inherit several methods the most important of which is the *Subscribe* method. This method allows Notifiable objects to subscribe to the primitive events generated by instances of subclasses of the Reactive class. Once this subscription takes place, the notifiable object will be informed of the primitive events generated by the Reactive object. For example, if X is a Reactive object and Y is a Notifiable object, then Y will be informed of the primitive events generated by X after the statement **X.Subscribe(Y)** is executed.

Notifiable class. Similarly, the public interface of the Notifiable class consists of methods which allow objects to receive and record primitive events generated by reactive objects. For an object to be notifiable it must be an instance of a class derived from the Notifiable class, i.e., an instance of a subclass of the Notifiable class. The method Record defined in this class documents the parameters computed when an event is raised, namely, the oid of the reactive object generating the event, the event generated, the time-stamp of when the event was generated, and the number and actual values of the parameters sent to the reactive object.

Event class hierarchy. The Event class is the superclass of an event class hierarchy which defines the common structure and behavior shared by all event types. Each event type is a subclass of the Event class. The event types that are supported are primitive as well as complex. The Primitive subclass is for modeling primitive events which are basically method invocations. Creation of a primitive event object requires indicating the *method* which raises the event and *when* the event should be raised, i.e., before or after execution of the method.

Rule class. The primary structure defining a rule is the event which triggers the rule, the condition which is evaluated when the rule is triggered, and the action which is executed when the rule is triggered. Therefore, creation of a rule object X is accomplished by executing the statement **Rule X(eventid, Condition, Action)**, where eventid is the oid

[1]Another way a class can become a reactive class is if it is a friend class of another reactive class.

of the event object representing the event that triggers the rule X, Condition is a function that is to be executed when the event is triggered and Action is a function to be executed if the Condition function returns true.

6.3 Adequacy of Sentinel for Supporting Transaction Models

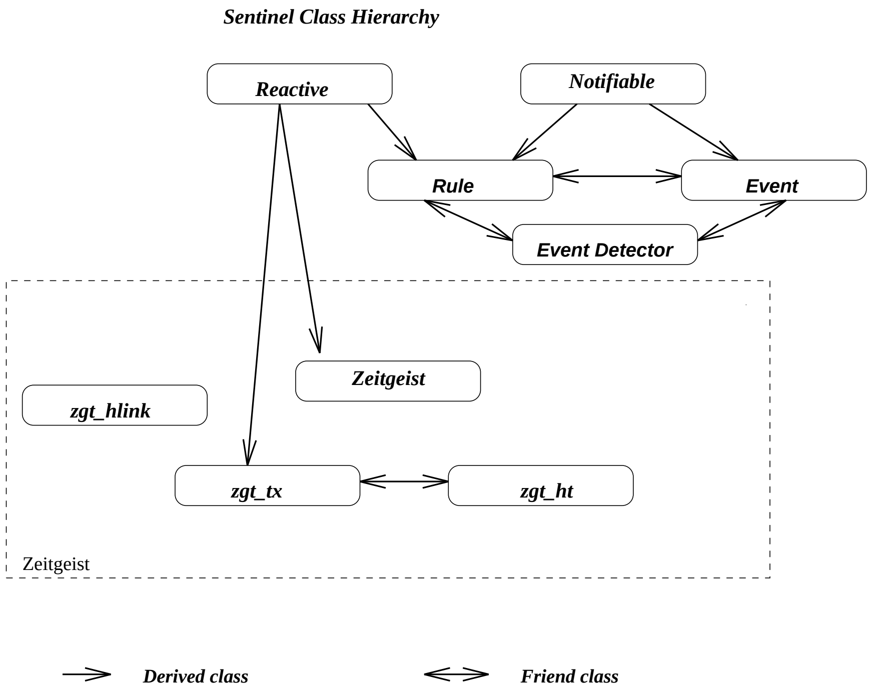


Figure 6.8. Sentinel class hierarchy at the System Level

While first incorporating active behavior into Zeitgeist we aimed at achieving two primary goals. The first goal was to provide a seamless incorporation of ECA rules into the OO paradigm while the second was use of this active capability at the *application level* to enforce integrity constraints as well as achieve additional application level functionality. These two main goals were met by treating events and rules as first class objects as well as creating a class hierarchy which provided active functionality. As the main intent for using active capability was to enforce application level constraints and integrity constraints, active behavior was incorporated *only* at the application level. In other words, only application level classes were derived or alternately made subclasses of the *Reactive class* as shown in Figure 6.7. Now, however, we require the trapping of *system level* operations as opposed to application level operations. This is easily achieved in Sentinel due to the versatility of its design. In particular, due to Sentinel providing the ability to trap operations in *one* centralized place, namely, the *Reactive* class, all that is required to trap system level operations is to derive the pertinent classes from the *Reactive class*. Consequently, since we are only interested in the operations pertinent to transactions and the TM, we derived the *zeitgeist* class and the *zgt_tx* class from the *Reactive* class. This automatically now allows one to define rules to be executed when methods of these classes are invoked. Thus we modified the Zeitgeist class hierarchy as illustrated in Figure 6.8.

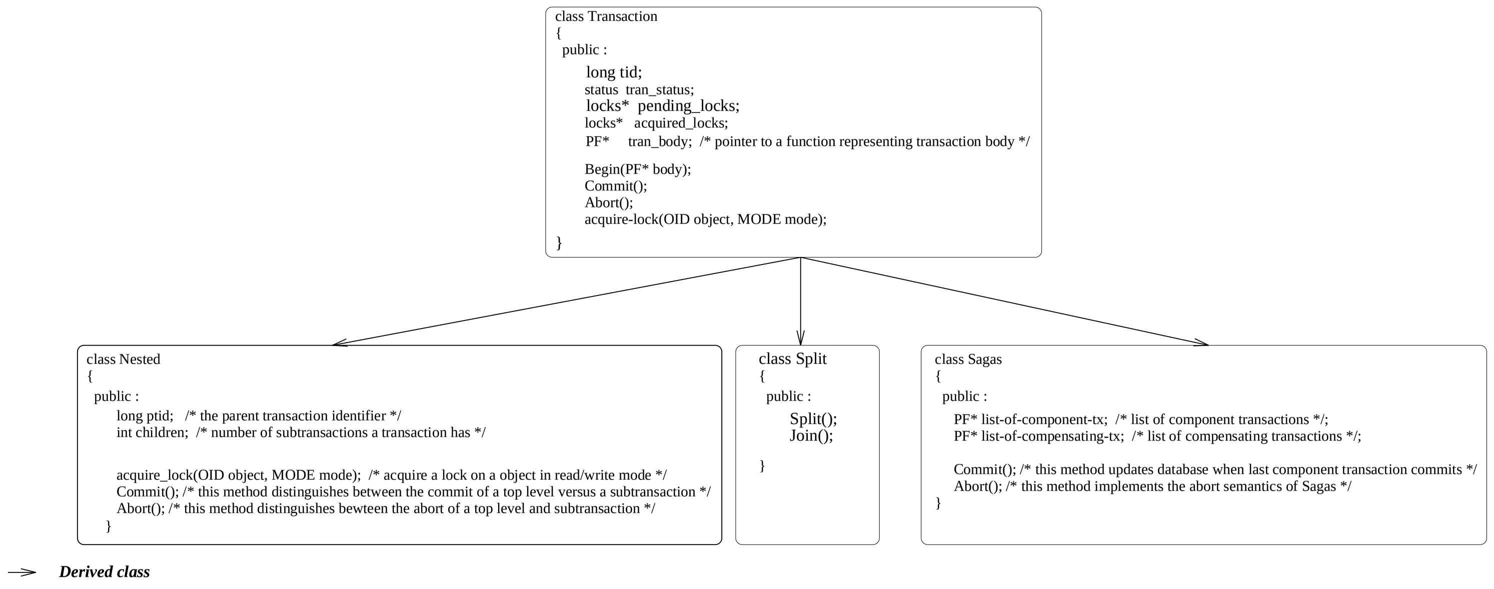
Alternatives for Supporting Transactions in the OO Paradigm

In this section we examine the alternatives for supporting various transaction models on the same DBMS and provide the rationale for our design choices. This discussion is in the context of the object-oriented paradigm. Before we begin, it is necessary to point out that we treat transactions as first class objects. The rationale for this design choice is that transactions exhibit the same properties as other objects, namely, transactions have a state (e.g., running, suspended, aborting), a structure (e.g., statements constituting transaction body, transaction identifier etc.), and behavior (a set of methods such as begin, commit defining its interface). The two main alternatives for supporting various transaction models are:

1. Create a transaction hierarchy (or alternatively a class hierarchy) where each class definition in the hierarchy models the required semantics of a particular transaction model. A sketch of a possible class hierarchy is depicted in Figure 6.9. This approach is motivated by runtime processing gains, since the semantics of a transaction is hard-wired into the DBMS and no additional processing at runtime is required. Although this approach naturally fits into the object-oriented paradigm, it suffers from several

limitations, primarily inextensibility. More specifically, supporting newer transaction models warrants modification of the existing transaction hierarchy and thus recompilation of the system. This requires the end user to be familiar with the code pertaining to the transactional system as well as wasting time while it is being recompiled. More importantly, once a transaction is created, its transaction semantics is determined (depending on the class it is an instance of) and cannot be changed. This may be a severe limitation when the user is unsure about which transaction model most adequately suits an application's semantics or when the user wants to perform some experimentation with different transaction models.

2. Create only one transaction class, as opposed to a class hierarchy, which contains all the methods such as begin, commit and abort that are necessary for modeling the different transaction models. However, since the semantics of these methods differ from one transaction model to the other, these methods should serve no purpose other than notifying when these methods are executed by transactions, i.e., invocation of these methods generate events. These generated events are subsequently propagated to the set of rules associated with a transaction and the condition and action executed for those rules which are triggered. This approach clearly introduces a runtime performance penalty incurred as a result of event trapping, condition evaluation and possible action execution. However, the advantages of this approach, namely, the ability to model existing, newer and arbitrary transaction semantics without modifying the underlying DBMS, outweigh this performance penalty. This is the approach adopted in our implementation.



```
class Transaction
{
  public :
        long tid;
        status  tran_status;
        locks*  pending_locks;
        locks*   acquired_locks;
        PF*     tran_body;  /* pointer to a function representing transaction body */

        Begin(PF* body);
        Commit();
        Abort();
        acquire-lock(OID object, MODE mode);
}
```

```
class Nested
{
  public :
        long ptid;   /* the parent transaction identifier */
        int children;  /* number of subtransactions a transaction has */


        acquire_lock(OID object, MODE mode);  /* acquire a lock on a object in read/write mode */
        Commit(); /* this method distinguishes between the commit of a top level versus a subtransaction */
        Abort(); /* this method distinguishes bewteen the abort of a top level and subtransaction */
    }
```

```
class Split
{
  public :
        Split();
        Join();

}
```

```
class Sagas
{
  public :
     PF* list-of-component-tx;  /* list of component transactions */;
     PF* list-of-compensating-tx;  /* list of compensating transactions */;

     Commit(); /* this method updates database when last component transaction commits */
     Abort(); /* this method implements the abort semantics of Sagas */
}
```

Figure 6.9. Transaction class hierarchy.

CHAPTER 7
IMPLEMENTATION DETAILS

In this chapter we give the details of implementing nested transaction, Split transactions and Sagas using the active database paradigm. ECA rules which formulate the semantics of the above mentioned transaction models are given in following sections.

7.1 Modeling Nested Transactions

In the nested transaction model [35], a transaction may contain any number of subtransactions, and each subtransaction, in turn, may contain any number of subtransactions. Hence, the entire transaction forms a hierarchy of transactions the root of which is called the *root* or *top-level* transaction. Transactions having subtransactions are called *parents*, and their subtransactions are their *children*. The transactions on the path from a transaction to the root of the transaction tree are called the *superiors* of the transaction. The nested transaction model allows several types of concurrency: sibling concurrency, parent/child concurrency, and the most general case – complete concurrency. We focus on sibling concurrency as it is the most widely used nested transaction model.

With respect to transaction semantics, top-level transactions have all the properties of traditional transactions. That is, top-level transactions preserve the ACID properties. Nested transactions preserve serializability among subtransactions; therefore, subtransactions cannot cooperate or share data. The commit of a subtransaction is conditionally subject to the commit of its superiors. Hence, a subtransaction's updates become permanent only when the enclosing top-level transaction commits. Upon commit, all locks held by a subtransaction are inherited by the parent transaction. A parent transaction does not interfere with its children (in sibling concurrency); a transaction is allowed to hold a lock if the conflicting transaction is one of its superiors.

The following rule, **Tx_initiate**, initiates both top-level and nested transactions by placing them on the scheduler queue. This rule is triggered when the *begin* method is invoked, i.e., when the begin event is *raised*.

Rule: Tx_initiate

On `T1->Transaction_Descriptor::begin()` *// detecting invocation of begin method*

Condition `True` *// no condition checking necessary*

Action `sched->Scheduler::Insert(T1->tid)` *// Place transaction on scheduler queue*

The next rule, **Tx_Release_All_Locks**, releases all locks held by a transaction. The chained list of locks held is traversed and each lock released to the outside world, i.e., a conventional release. This rule is triggered by the execution of other rules, specifically, *Tx_commit_TopLevel* and *Tx_abort_TopLevel*. This exhibits how it is possible to exploit the cascading of rule execution to modularize rules. In other words, it is possible to create rules which perform common operations and use these rules in more than one transaction model.

Rule: Tx_Release_All_Locks

On `T1->Transaction_Descriptor::release_locks()` *// detecting release_locks method*

Condition `True` *// no condition checking necessary*

Action *// start releasing all locks held*

```
      trav = T1->locks;      // point to head of lock list held by T1
      while(trav != NULL) {
         get_exclusive_sem(); // get exclusive semaphore to access shared data
          trav->ODO->counter--; // decrement no. of transactions pointing to ODO
          temp = trav->ODO;      // make a temporary variable point to ODO
          trav = trav->next_obj_desc; // traverse to next lock held by transaction
```

```
        if(temp->counter == 0) // check if no transactions pointing to ODO
          free(temp);      // release memory used by ODO

        release_exclusive_sem();      // release exclusive semaphore
    }
    T1->locks = NULL;      // set T1's lock list to NULL
```

Rules **Tx_commit_TopLevel** and **Tx_commit_Child**, defined below, are triggered by the same event, namely, *commit* of a transaction. These two rules capture the difference in commit semantics between top-level and nested transactions. Note that the rules **Tx_commit_TopLevel** and **Tx_commit_Child** can be further simplified by introducing a new event such as *delegate* which will delegate the locks to the enclosing transaction. In the case of the top-level transaction, the enclosing transaction will be the outside world (i.e., a regular release) and for nested transactions the release will be to the immediate superior. Also note that rule **Tx_commit_TopLevel** triggers rule **Tx_Release_All_Locks** given above.

Rule: Tx_commit_TopLevel

```
On T1->Transaction_Descriptor::commit()      // detecting invocation of method commit
Condition T1->transaction_type == TOPLEVEL      // T1 is a top-level transaction
Action
      Make updates permanent      //based on recovery method used
      Raise release_locks event      //this triggers rule Tx_Release_All_Locks
```

Rule: Tx_commit_Child

```
On T1->Transaction_Descriptor::commit()      //detecting invocation of method commit
```

```
Condition T1->transaction_type == CHILD      // T1 is not a top-level transaction
Action
// delegate operations on shared objects to parent as well as release locks to parent
    trav = T1->locks;      // point to head of lock list held by T1
    while(trav != NULL)
    {
        delegate operations performed on this object to Parent(T1);
        // T1's parent is now responsible for these operations
        if(trav->ODO->oid ∈ list of objects held by Parent(T1)) // check if lock
                                                                  also held by parent
        {
            get_exclusive_sem(); // get an exclusive semaphore to access shared data
            set parent's lock to most exclusive lock held by parent & child
            release_exclusive_sem();      // release exclusive semaphore
        }
        else
        {
            get_exclusive_sem();      // get an exclusive semaphore
            change lock-mode to RETAINED      // for parent to inherit lock
            add this ODO to parent(T1)'s list // Parent(T1) now points to this object
            release_exclusive_sem();      // release exclusive semaphore
        }
    }
    T1->locks = NULL;      // set T1's lock list to NULL
```

Similarly, rules **Tx_abort_TopLevel** and **Tx_abort_Child** describe the semantics of *abort* for top-level and nested transactions. Again, they are triggered by the raising of the same event, namely, *abort*.

Rule: Tx_abort_TopLevel

On `T1->Transaction_Descriptor::abort()` *// detecting invocation of method abort*

Condition `T1->transaction_type == TOPLEVEL` *// T1 is a top-level transaction*

Action

```
    Flush buffers
    Raise release_locks event
```

// discard all changes made to objects (Flush buffers)
//this triggers rule Tx_Release_All_Locks (Raise release_locks event)

Rule: Tx_abort_Child

On `T1->Transaction_Descriptor::abort()` *// detecting invocation of method abort*

Condition `T1->transaction_type == CHILD` *// T1 is a child transaction*

Action

`Flush buffers` *// discard all changes made to objects*

`Raise release_locks event` *//this triggers rule Tx_Release_All_Locks*

In the rest of this section, we show rules defining lock acquisition semantics for top-level and child transactions. These rules are **Tx_acquire_exclusive_lock_TopLevel** and **Tx_acquire_lock_Child**. Both these rules use the **Tx_grant_lock** rule which basically updates the transaction and lock table to reflect lock acquisition. Rules for acquiring shared and exclusive locks for top-level transactions are also given below.

Rule **Tx_grant_lock** creates a new entry in the object hash table when a transaction acquires a lock on an object. All necessary updates to the transaction and object table are performed by this rule.

Rule: Tx_grant_lock

On `T1->Transaction_Descriptor::acquire_lock(oid,mode)` *// detecting add_lock method*

Condition `True` *// no condition checking necessary*

Action

// Create a new ODO, insert it in object hash table, & add it to list of locks held by T1

```
    Object_Descriptor* ODO(oid,mode);     // create new ODO
    i = hash(oid);     // find bucket to insert new ODO
    get_exclusive_sem();     // get an exclusive semaphore to modify object hash table
    object_table[i].insert(ODO);     // insert ODO in bucket
    Make T1 point to new ODO in bucket     // insert ODO at end of lock list held by T1
    release_exclusive_sem();     // release exclusive semaphore
```

Rule **Tx_acquire_exclusive_lock_TopLevel** defines the semantics of *exclusive* lock acquisition for top-level transactions. Once **Tx_acquire_exclusive_lock_TopLevel** is triggered, we first check whether the transaction already holds the lock in the requested mode. If this is the case, then no action is performed and the transaction simply proceeds with its execution. Otherwise, we check whether the lock is held in a conflicting mode. If this is found to be true, then the transaction is blocked on a semaphore until the lock is released. If the lock is available, it is granted and the transaction proceeds with its execution.

Rule: Tx_acquire_exclusive_lock_TopLevel

On `T1->Transaction_Descriptor::acquire_lock(oid,mode)` *// detecting invocation of method acquire_lock*

Condition

// TOPLEVEL & lockmode is X-mode & no transaction holds lock in X- or S-mode

```
       if(T1->transaction_type != TOPLEVEL || mode != EXCLUSIVE)
           return(0);
```

```
    if T1 already holds lock in EXCLUSIVE mode
    {
       found = 1;      // flag indicating that transaction already holds lock
       return(1);
    }
    i = hash(oid);      // hash object wanted to find bucket
    trav = object_table[i];      // point to head of list of bucket
    while(trav != NULL)      // start looping through objects in bucket
    {
       if(trav->oid != oid)      // check if object is in hash table
        {
          trav = trav->next_in_bucket;      // move to next object in bucket
          continue;      // go to top of while loop
        }
       Block transaction on semaphore      // object is in hash table, i.e., held in
                                           X- or S-mode and transaction must wait
       break;      // break out of loop and acquire lock once transaction is unblocked
    }

    return(1);
```

Action

```
  if(!found)      // if transaction does not already hold the lock
    Raise grant_lock event      // this triggers rule Tx_grant_lock
```

Rule **Tx_acquire_lock_Child** defines the semantics of lock acquisition in all modes for nested transactions. A transaction is allowed to proceed if it already holds the lock in the requested mode or the transaction holding the lock in conflicting mode is an ancestor transaction. Otherwise, the subtransaction is blocked on a semaphore until it can acquire the lock.

Rule: Tx_acquire_lock_Child

On `T1->Transaction_Descriptor::acquire_lock(oid,mode)` *// detecting invocation of method acquire_lock*

Condition // check transaction type & locking rules

```
if(T1->transaction_type != CHILD)      // T1 is a child transaction
    return(0);
if T1 already holds the lock in required mode
{
    found = 1;      // flag indicating that transaction already holds lock
    return(1);
}

SUPERIORS = T1's superior transactions;
if(mode == EXCLUSIVE) || mode == READONLY)) // mode is X- or RO-mode
    TS = set of transactions holding either an X- or S-lock on oid
else      // requested mode is S-mode
    TS = set of transactions holding an X-lock on oid
$\forall$ $t_i$ such that $t_i \in$ TS
    if $t_i \notin$ SUPERIORS
        block transaction on semaphore;
    return(1)
```

Action

```
   if(!found)        // if transaction does not already hold the lock
     Raise grant_lock event        // this triggers rule Tx_grant_lock
```

Tx_acquire_shared_lock_TopLevel and **Tx_acquire_readonly_lock_TopLevel** are two additional rules given below. These rules complete the semantics of lock acquisition for top-level transactions. **Tx_acquire_shared_lock_TopLevel** defines the semantics of shared lock acquisition while **Tx_acquire_readonly_lock_TopLevel** defines the semantics of read-only acquisition. These rules allow a transaction to continue executing if it already holds the lock or if no transaction holds the lock in a conflicting mode. If the lock is held in a conflicting mode, then the transaction is blocked until it can be granted the lock.

Rule: Tx_acquire_shared_lock_TopLevel

On `T1->Transaction_Descriptor::acquire_lock(oid,mode)` *// detecting invocation of method acquire_lock*

Condition *// TOPLEVEL & lockmode is S-mode & no transaction holds lock in X-mode*

```
    if(T1->transaction_type != TOPLEVEL || mode != SHARED)
       return(0);
    if T1 already holds lock in SHARED mode {
       found = 1;      // flag indicating that transaction already holds lock
       return(1);
    }
    i = hash(oid);      // hash object wanted to find bucket
    trav = object_table[i];       // point to head of list of bucket
    while(trav != NULL) {       // start looping through objects in bucket
    // check if object is in hash table
```

```
            if( (trav->oid != oid) ||
                (trav->oid == oid && trav->lock_mode != EXCLUSIVE)) {
              trav = trav->next_in_bucket;      // move to next object in bucket
              continue;       // go to top of while loop
            }
            Block transaction on semaphore;      // object is held in X-mode
            break;       // break out of loop & acquire lock once transaction is unblocked
         }

         return(1)
```

Action

```
       if(!found)       // if transaction does not already hold the lock
         Raise grant_lock event       // trigger rule Tx_grant_lock
```

Rule: Tx_acquire_readonly_lock_TopLevel

On `T1->Transaction_Descriptor::acquire_lock(oid,mode)` *// detecting invocation of method acquire_lock*

Condition

// TOPLEVEL & lockmode is RO-mode & no transaction holds lock in X- or S-mode

```
         if(T1->transaction_type != TOPLEVEL || mode != READONLY)
            return(0);

         if T1 already holds lock in READONLY mode
         {
```

```
    found = 1;      // flag indicating that transaction already holds lock
    return(1);
  }
  i = hash(oid);      // hash object wanted to find bucket
  trav = object_table[i];      // point to head of list of bucket
  while(trav != NULL)      // start looping through objects in bucket
  {
    if(trav->oid != oid)      // check if object is in hash table
    {
      trav = trav->next_in_bucket;      // move to next object in bucket
      continue;      // go to top of while loop
    }
    Block transaction on semaphore    // object already held in X- or S-mode
    break;      // break out of loop & acquire lock once transaction is unblocked
  }

  return(1);
```

Action

```
  if(!found)      // if transaction does not already hold the lock
    Raise grant_lock event      // trigger rule Tx_grant_lock
```

7.2 Split Transactions

Split transactions [40] were proposed mainly for supporting open-ended applications. In this transaction model, a transaction can execute the operation *split-transaction* which basically creates a *new* top-level transaction. The original transaction and the new transaction are serialized as if they are two independent transactions. However, when the original transaction executes the operation *split-transaction*, it can delegate responsibility of uncommitted operations on a *specific* subset of objects to the newly created transaction. After the split occurs, the two transactions continue execution and commit or abort independently. Similarly, a transaction can also execute the operation *join-transaction* which essentially combines two active serializable transactions into one transaction. The main advantage of split transactions is relaxing isolation which is achieved when either the original or new transaction commits and releases its results.

One approach for supporting split transactions is to write ECA rules expressing their semantics using the three step process described in section 5.2.1. That is, identify the events, write new sets of conditions/actions, and combine them into rules. Although this approach yields a correct solution, it does not exploit reusability of rules among transaction models. A more beneficial approach is to examine currently defined ECA rules (i.e., events, conditions and actions defined for supporting various transaction models) and determine their reuse (either entire rules or components thereof) for expressing the new transaction model at hand. This allows one to understand the similarities as well as differences among transaction models, expedite the definition of the semantics of a transaction model (as rules may no longer need to be written from scratch), reduce the number of rules in the system, and most importantly provide extensibility.

The latter approach was adopted for defining the rules necessary for supporting split transactions. Transactions belonging to this model are essentially top-level transactions exhibiting the same semantics as top-level transactions in the nested transaction model. Therefore, the semantics of *begin*, *commit*, *abort* and *acquire_lock* are identical to the semantics of these operations in top-level transactions of the nested transaction model. Consequently, the rules defined for these methods, given in section 7.1, are also applicable to this transaction model. The operations *split-transaction* and *join-transaction* are specific to this model and thus ECA rules realizing their semantics need to be defined. These rules are given below.

Rule **Tx_split** is triggered when a transaction invokes the *split* operation. This rule creates a new top-level transaction and delegates the locks and uncommitted operations on the indicated objects to the new transaction.

Rule: Tx_split

```
On T1->Transaction_Descriptor::split(obj_set, tx_body)      // detecting invocation
                                                               of split method
Condition True          // no condition checking necessary
Action
    Transaction_Descriptor *New_Tx;      // create new transaction descriptor object
    tid = New_Tx->Create_Descriptor(tx_body, TOPLEVEL); // initialize & get tid of
                                                          new toplevel transaction
    ∀ $o_i$ such that $o_i$ ∈ obj_set
        delegate($o_i$, tid);      // delegate all locks & uncommitted
                                      operations on $o_i$ to tid
    sched->Scheduler::Insert(T1->tid)    // Place new transaction on scheduler queue
```

Rule **Tx_join** is triggered when a transaction invokes the *join* operation. This rule first checks that the transaction to be joined with is active. If this is found to be true both transactions are combined into one top-level transaction.

Rule: Tx_join

```
On T1->Transaction_Descriptor::join(Tx)      // detecting invocation of join method
Condition Tx->transaction_status == ACTIVE      // transaction to join with is active
```

Action

```
// Delegate objects to the transaction to be joined with
OBJECTS = set of objects held by T1
∀ $o_i$ such that $o_i \in$ OBJECTS
    delegate($o_i$, Tx->tid);      // delegate all locks & uncommitted operations
                                   on $o_i$ to Tx
T1->~Transaction_Descriptor();         // remove transaction issuing join
                                       operation from transaction table
```

7.3 Sagas

Sagas [23] is a transaction model introduced to more adequately serve the requirements of long-lived transactions. A Saga consists of a set of independent component transactions $T_1, T_2, ..., T_n$ where each component transaction T_i (except transaction T_n) has an associated compensating transaction CT_i. Compensating transactions *semantically undo* the effects of their respective component transaction. The component transactions $T_1, T_2, ..., T_n$ execute serially in a predefined order and may interleave arbitrarily with the component transactions of other sagas. If a component transaction aborts, then the entire Saga aborts by executing the compensating transactions in reverse order to the order of the commitment of the component transactions. Here, we do not show the ECA rules which define the semantics of this transaction, but rather discuss how to modify the ECA rules defining nested transactions to achieve the semantics of this model.

Component and compensating transactions are top-level transactions whose semantics for *begin*, *commit*, *abort* and *acquire_lock* are very similar to those of top-level transactions belonging to the nested transaction model. To elaborate, the semantics of *commit* in a top-level component transaction is to make all updates permanent to the database and release all locks held. This is precisely the semantics of commit by a top-level transaction in the nested transaction model. In addition, the commit of a component transaction should also begin executing the next component transaction in the series. Therefore, the action part of rule **Tx_commit_TopLevel** should be modified to include starting the execution of the next component in the saga series. Similarly, the abort of a component transaction performs the same operations as the abort of a top-level transaction in the nested transaction model. In addition, it also starts execution of the appropriate compensating transaction in order to start the rollback process. Thus the action part of rule **Tx_abort_TopLevel**, given in section 7.1, should be modified to reflect this difference.

Likewise, the *commit* of a compensating transaction performs all operations carried out by the commit of a top-level transaction in the nested transaction model. In addition, it should also start the execution of the next compensating transaction in the rollback process. Similarly, the abort of a compensating transaction performs all operations executed when a top-level transaction aborts. However, it also restarts the compensating transaction again, i.e., the aborted compensating transaction is retried until it successfully commits.

CHAPTER 8
EXTENSIBILITY

In this chapter we discuss the extensibility aspects of our approach. There are two distinct aspects of extensibility that need to be addressed: i) extensibility of ECA rules as compared to other approaches (object-oriented and tool-kit) to extensibility and ii) extensibility in modeling newer transaction models. Below, we address each of the above.

We believe that ECA rules at the system level provide yet another, but more powerful form of extensibility. In contrast to the other two approaches (object-oriented and tool-kit), this approach provides greater control at runtime (with respect to the object-oriented approach) and allows one to redefine semantics dynamically. In a sense, the binding of rules can be controlled by other rules instead of overloading which provides a fixed form of dynamic association.

In contrast to the tool-kit approach, use of rules allows one to support both application-level and system-level modification of behavior in a uniform manner. Further, our approach does not preclude the inline incorporation/compilation of rules to avoid the performance overhead that is associated with rule processing. However, the use of rules allows one to modularize and prototype systems relatively easily.

So far we have used ECA rules defined at the system level to achieve the semantics of various transaction models. The rule sets defined for the various transaction models focus on the concurrency control aspect of transaction models. Since Zeitgeist uses a lock based method for achieving concurrency control, we also adopted this method in our rules. In particular, we defined events for the operations such as *lock-acquisition*, *lock-release* and *upgrade-lock*. Another reason which prompted our use of a lock-based mechanism for concurrency control, is that it is used in most commercial DBMSs, is well understood and is perhaps the most popular of the concurrency control mechanisms. However, it is important to realize that our approach to realizing transaction models is not limited to a particular concurrency control method. Rather, our approach is extensible enough to be applied to other concurrency control mechanisms, e.g., optimistic concurrency control (OCC).

To show the extensibility of our approach let us assume that OCC using timestamp ordering is preferred over a lock based method. The basic notion behind OCC is to allow transactions to read, compute, and update local copies freely without updating the actual database. Some information is maintained with each data item to ensure serializability of committed transactions. Once a transaction completes, it enters a validation phase which consists of checking if the updates maintain the consistency of the database (i.e., the commit of the transaction is serializable). If the answer is affirmative, then the updates are made persistent in the database, otherwise the transaction is aborted.

The three rules of the OCC algorithm [30] using read and write sets can be translated into ECA rules when the commit is issued by a transaction. In Zeitgeist, only the object-id is kept in the shared memory data structures (along with some other information, but not the value). Local copies of the objects are maintained in the application/client address space. By modifying the tables in the shared memory to keep the timestamp information, it is relatively easy to implement the OCC algorithm based on timestamp order by writing rules on the commit and disabling rules on acquire lock etc. The list of objects accessed by a transaction is already maintained (although there is no distinction between read and write objects) in shared memory.

As previously mentioned, we have concentrated on using ECA rules for the concurrency aspects of transaction models. This, however, does not prohibit its utilization to other aspects of transaction management. Consequently, our approach can also be used for the recovery aspects of transaction management as well as other aspects such as deadlock detection and deadlock resolution. Usage of this paradigm in these other areas of transaction management entails identifying the data structures and operations that need to be detected or trapped as well as defining the semantics of the operations using ECA rules. Therefore, the exact same process used for supporting the concurrency control aspects is utilized to other aspects of transaction management.

CHAPTER 9
CONCLUSIONS AND FUTURE WORK

Although database systems encompass a large range of applications, they were originally developed for business oriented database applications such as banking systems, airline reservation systems, and organizational systems. These applications are well-served by the properties of the traditional transaction model, primarily due to the fact that these applications are simple in nature and have very short duration. However, as the scope of databases extends to a large variety of applications, such as workflow management, cooperative tasks, and computer integrated manufacturing (CIM), it is important to reevaluate the assumptions and properties of the traditional model of transactions. These applications expose the inadequacy of the traditional transaction model in meeting the requirements of these non-traditional applications. For instance, the failure atomicity requirement of the traditional transaction model dictates that all work must be rolled back in the event of failures. This requirement is unsuitable for long-lived transactions due to the fact that much work might have been done and will be lost in the event of a failure. In addition, the traditional transaction model does not allow much cooperation among activities. Cooperation is required in CAD environments where several people may be jointly working on a project where each person is responsible for part of the design project.

The current solution to meeting the diverse requirements of non-traditional applications has been the proposal of a number of advanced or extended transaction models such as nested transactions, Sagas, ConTract model, and Flex transaction model. These transaction models relax the ACID properties in various ways to better model the parallelism, consistency, and serializability requirements of non-traditional applications. Proposals of advanced transaction models primarily start from a specific application. To elaborate, an application's dynamic behavior is analyzed, a fault model is specified, and features are either added or modified to the classical ACID transaction model aiming at supporting the requirements of that application. Although a large number of transaction models have been proposed, little effort has been made in implementing them and understanding the interaction among different transaction models.

The goal of this research is to provide a framework which allows for the specification and enforcement of different transaction models on *any one* given DBMS. We have proposed to use the active database paradigm as a mechanism for the specification and enforcement of extended transaction models on a DBMS. We first identified the main alternatives for supporting various transaction models given a DBMS and examined the advantages and disadvantages of each. Next, we examined the semantics of various transaction models and translated them into high-level specifications using ECA rules. We then chose to specify and enforce various transaction models using ECA rules defined on *system level* events such as operations on the transaction table, the lock table and the log. We identified the main data structures and operations on them that need to be implemented in order to support extended transactions. The class definitions (in C++) were written to define these data structures and operations. We then examined the adequacy of Sentinel, an active OODBMS, for supporting extended transactions and showed how to modify its class hierarchy to achieve this. Several extended transaction models were examined and their semantics translated into ECA rules defined on the data structures mentioned above. We also implemented the traditional transaction model as well as Sagas using the active database paradigm. Sentinel is the active OODBMS used for our implementation.

REFERENCES

[1] E. Anwar. Supporting complex events and rules in an oodbms: A seamless approach. Master's thesis, Database Systems R&D Center, CIS Department, University of Florida, E470-CSE, Gainesville, FL 32611, November 1992.

[2] E. Anwar, L. Maugis, and S. Chakravarthy. A New Perspective on Rule Support for Object-Oriented Databases. In *Proceedings, International Conference on Management of Data*, pages 99–108, Washington, D.C., May 1993.

[3] P. Attie, M. Singh, M. Rusinkiewicz, and A. Sheth. Specifying and enforcing intertask dependencies. Technical Report MCC Report: Carnot-245-92, Microelectronica and Computer Technology Corporation, November 1992.

[4] R. Badani. Nested Transactions for Concurrent Execution of Rules: Design and Implementation. Master's thesis, Database Systems R&D Center, CIS Department, University of Florida, Gainesville, FL 32611, October 1993.

[5] A. Biliris, S. Dar, N. Gehani, H.V.Jagadish, and K. Ramamritham. ASSET: A System for Supporting Extended Transactions. In *Proceedings, International Conference on Management of Data*, May 1994.

[6] A. Biliris, S. Dar, N. Gehani, H. V. Jagadish, and K. Ramamritham. ASSET: A System for Supporting Extended Transactions. In *Proceedings, International Conference on Management of Data*, pages 44–54, Minneapolis, Minnesota, May 1994.

[7] J. A. Blakeley, C. W. Thompson, and A. M. Alashqur. Oql[x] : Extending a programming language x with a query capability. Technical Report TR 90-07-01, Texas Instruments, July 1990.

[8] J. A. Blakeley, C. W. Thompson, and A. M. Alashqur. Strawman reference for object query languages. *Proceedings of the First OODB Standardization Workshop*, May 1990.

[9] J. A. Blakeley, C. W. Thompson, and A. M. Alashqur. Zeitgest query language (zql). Technical Report TR-90-03-01, Texas Instruments, March 1990.

[10] Jose A. Blakeley. Open Object Database Management Systems. In *Proceedings, International Conference on Management of Data*, page 520, Minneapolis, Minnesota, May 1994.

[11] S. Chakravarthy. Active Database Management Systems: Requirements, State-Of-The-Art, and an Evaluation. In H. Kangassalo, editor, *Entity-Relationship Approach: The Core of Conceptual Modeling*, pages 461–473. Elsevier Science Publishers, North-Holland, 1991.

[12] S. Chakravarthy and R. Blanco-Mora. Supporting very large production systems using active dbms abstraction. Technical Report UF-CIS TR-91-25, Database Systems R&D Center, CIS Department, University of Florida, E470-CSE, Gainesville, FL 32611, Sep. 1991.

[13] S. Chakravarthy and S. Garg. Extended relational algebra (era): for optimizing situations in active databases. Technical Report UF-CIS TR-91-24, Database Systems R&D Center, CIS Department, University of Florida, E470-CSE, Gainesville, FL 32611, Nov. 1991.

[14] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim. Composite Events for Active Databases: Semantics, Contexts, and Detection. In *Proceedings, International Conference on Very Large Data Bases*, pages 606–617, August 1994.

[15] S. Chakravarthy, V. Krishnaprasad, Z. Tamizuddin, and R. Badani. ECA Rule Integration into an OODBMS: Architecture and Implementation. Technical Report UF-CIS-TR-94-023, University of Florida, E470-CSE, Gainesville, FL 32611, Feb. 1994. (In ICDE-95, Taiwan, March 1995.).

[16] S. Chakravarthy and D. Mishra. An event specification language (snoop) for active databases and its detection. Technical Report UF-CIS TR-91-23, University of Florida, E470-CSE, Gainesville, FL 32611, Sep. 1991.

[17] S. Chakravarthy and D. Mishra. Snoop: An Expressive Event Specification Language for Active Databases. *Data and Knowledge Engineering*, 14(10):1–26, October 1994.

[18] P. K. Chrysanthis and K. Ramamtitham. Acta: A framework for specifying and reasoning about transaction structure and behavior. In *Proceedings, International Conference on Management of Data*, pages 194–203, 1990.

[19] F. Eliassen, J. Veijalainen, , and H Tirri. Aspects of transaction modelling for interoperable information systems. in interim report of the cost 11ter project,, 1988.

[20] A. Elmagarmid, Y. Leu, W. Litwin, , and M. Rusinkiewicz. A multidatabase transaction model for interbase. In *Proceedings, International Conference on Very Large Data Bases*, Brisbane,Australia, 1990.

[21] A. Elmagarmid, Y. Leu, W. Litwin, and M. Rusinkiewicz. A multidatabase transaction model for Interbase. In *Proceedings of International Conference of Very Large Data Bases*, August 1990.

[22] A.K. Elmagarmid, editor. *Database Transaction Models for Advanced Applications*. Morgan Kaufmann Publishers, San mateo, CA, 1992.

[23] H. Garcia-Molina and K. Salem. Sagas. In *Proceedings of the Conference on Database Systems in Office, Technique and Science*, pages 249–259, May 1987.

[24] D. Georgakopoulos, M. Hornick, P. Krychniak, and F. Manola. Specification and management of extended transactions in a programmable transaction environment. In *Proceedings IEEE Conference on Data Engineering*, February 1994.

[25] Jim Gray. The transaction concept: Virtues and limitations. In *Proceedings, International Conference on Very Large Data Bases*, Cannes, France, 1981.

[26] T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys*, 1983.

[27] Sandra Heiler, Sara Haradhvala, Zdonik, Barbara Blaustein, and Aron Rosenthal. A flexible framework for transaction management in engineering environments. in database trasaction models for advanced applications, edited by a. elmagarmid,.

[28] Ishikawa. Object-Oriented Real-Time Language Design: Constructs for Timing Constraints. In *OOPSLA '90 proceedings*, pages 289–298, 1990.

[29] V. Krishnaprasad. Event Detection for Supporting Active Capability in an OODBMS: Semantics, Architecture, and Implementation. Master's thesis, Database Systems R&D Center, CIS Department, University of Florida, Gainesville, FL 32611, March 1994.

[30] H.T. Kung and J.T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2):213–226, 1981.

[31] F. Llirbat and E. Simon. Optimizing active database transactions: A new perspective. In *proc. of the 1st Int'l Workshop on Active and Real-Time Database Systems*, Skovde, Sweden, June 1995.

[32] L. Maugis. Adequacy of active oodbms to flight data processing servers. Master's thesis, National School of Civil Aviation / University of Florida, E470-CSE, Gainesville, FL 32611, August 1992.

[33] C. Mohan. Tutorial: A Survey and Critique of Advanced Transaction Models. In *Proceedings, International Conference on Management of Data*, page 521, Minneapolis, Minnesota, May 1994.

[34] M. Morgenstern. Active Databases as a Paradigm for Enhanced Computing Environments. In *Proceedings 9th International Conference on Very Large Data Bases*, pages 34–42, 1983.

[35] J. E. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing.* PhD thesis, Department of Electrical Engineering and Computer Science, MIT, 1981.

[36] M. Nodine and S. Zdonik. Cooperative transaction hierarchies: A transaction model to support design applications. In *Proceedings, International Conference on Very Large Data Bases*, pages 83–94, 1984.

[37] OODB. Open OODB Toolkit, Release 0.2 (Alpha) Document. Texas Instruments, Dallas, September 1993.

[38] K. Osterbye. Active Objects: An Access Oriented Framework for Object Oriented Language. In *JOOP*, pages 6–10, June/July 1988.

[39] M. T. Ozsu and P. Valduriez. *Principles of Distributed Database Systems.* Prentice-Hall International, Inc., Englewood Cliffs, N.J., 1991.

[40] C. Pu, G. Kaiser, and N. Hutchinson. Split-transactions for open-ended activities. In *Proceedings, International Conference on Very Large Data Bases*, 1988.

[41] A. Reuter. Contract: A means for extending control beyond transaction boundaries. In *Proceedings of the 2nd International Workshop on High Performance Transaction Systems*, September 1989.

[42] A. Reuter. Contracts: A means for extending control beyond transaction boundaries. *3rd INt'l Workshop on High Performance Transaction processing*, September 1989.

[43] Marek Rusinkiewicz and Amit Sheth. Polytransactions for managing interdependent data. ieee data engineering bulletin,, March 1991.

[44] A. Sharma. On extensions to a passive dbms to support active and multi-media capabilities. Master's thesis, CIS Department, University of Florida, Gainesville, 1992.

[45] Z. Tamizuddin. Rule Execution and Visualization in Active OODBMS. Master's thesis, Database Systems R&D Center, CIS Department, University of Florida, Gainesville, FL 32611, May 1994.

[46] Rainer Unland and Gunter Schlageter. *A Transaction Manager Development Facility for Non Standard Database Systems.*

[47] D. Wells, J. A. Blakeley, and C. W. Thompson. Architecture of an Open Object-Oriented Database Management System. *IEEE Computer*, 25(10):74–81, October 1992.

BIOGRAPHICAL SKETCH

Eman Anwar was born on April 25, 1969, in Cairo, Egypt. She received a Bachelor of Science in Computer Science degree from Kuwait University, Kuwait in January 1989. After her graduation, she worked as a computer programmer at the Institute of Banking Studies, Kuwait.

She joined the Department of Computer and Information Science and Engineering at the University of Florida in January 1991 to pursue a master's degree, and since then has worked as a research assistant in the Database Systems Research and Development Center of the department. She has also worked as a teaching assistant in the Computer and Information Science and Engineering Department of the university. Her research interests include computer networks, distributed operating systems, active database systems, and transaction processing.