ON EXTENSIONS TO A PASSIVE DBMS TO INCORPORATE ACTIVE AND MULTI-MEDIA CAPABILITIES

By

ASHEESH SHARMA

A THESIS PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

1992

Dedicated to my grandfather
Pt. R. L. Sharma,
a great advocate for justice

ACKNOWLEDGMENTS

Firstly, thanks are due to my adviser, Dr. Sharma Chakravarthy, for his continuous support, advice and guidance in this research. I am also thankful to Dr. Stanley Su and Dr. Herman Lam for serving on my committee and for their comments to improve the manuscript.

Mrs. Sharon Grant deserves special thanks for being so helpful in times of need. The never ending supply of coffee in the DBRC and the consistent reminders for the *coffee fund* made the stay memorable. I am also thankful to Mrs. Marlene Hughes for all the help and cooperation she provided.

On a personal note, I take this opportunity to thank all of the graduate students at this center for their help and friendship. Special thanks are due to Vijay and Neeta for their constant support and encouragement in the hours of distress during my graduate studies. I wish to express sincere appreciation to them for the inspiration they provided and for giving me a different outlook to life. The fruitful technical discussions with Arun Gopalan, who doubled as my workout partner, are worth mentioning. Many others, including Ajay, Ravishankar, Shashi, Taqi, Ravi, Niranjan, Chetan, Radha and Tarek, provided the much needed relief and company for many outdoor activities.

Last, but not the least, I thank my parents and brother for their love and support. Without their encouragement and guidance, this work would not have been possible.

This work is supported by NSF Grant IRI-9011216 and the Florida High Technology and Industrial Council Grant UPN# 90091013.

TABLE OF CONTENTS

Abstract of Thesis
Presented to the Graduate School of the University of Florida
in Partial Fulfillment of the Requirements for the
Degree of Master of Science

ON EXTENSIONS TO A PASSIVE DBMS TO INCORPORATE ACTIVE AND MULTI-MEDIA CAPABILITIES

By

Asheesh Sharma

August 1992

Chairman: Dr. Sharma Chakravarthy
Major Department: Computer and Information Sciences

Primitive active capabilities have been incorporated into databases quite early. Triggers have been defined quite explicitly in databases like System R, but the functionality of most of these systems has been limited to the enforcement of integrity constraints. Deductive data bases have been another area where the rules have been commercially incorporated and supported. These, however, cannot be considered active as they respond only to explicit queries from the outside world and hence are passive in the true sense.

This thesis represents work done to incorporate active capabilities into a passive DBMS. The work was divided into a design and implementation phase. The design part consisted of a nested transaction model for incorporating rules while the implementation was towards supporting multiple data representation. The design allows the support of different coupling modes in the future, and also provides a clean execution model for rule processing. The Ingres interface was done for supporting multiple Transactional Stores in the system. A storage Manager using Oracle as the Persistent

Storage was already existing at the original site. We interfaced the current system to use Ingres as the Storage Manager. This implementation also helped us to test the open OODB architecture. The final aim is to provide multiple object stores that can coexist. This kind of storage is useful for multimedia DBMSs where conventional data is stored as relations and images as long bit strings.

CHAPTER 1
INTRODUCTION

This thesis works towards extending the current database concepts for supporting some of the non-traditional database applications. These include the active database concept and also the multi-media capability.

Conventional DBMSs are said to be passive. This means that the database changes its state only on some user specified operation, either a query or some other update operation. This could also be application generated rather than user defined. We can say that **effect** is only by an explicit, external **cause**. Many ways have been suggested to make a database active. One is to have external modules interfacing with the DBMS to monitor events and conditions. We followed the integrated approach where event detection and condition monitoring are added to the existing passive DBMS. The schema definition as defined for the passive DBMS does not change, but specification of events and conditions to be monitored is added to the DBMS. These are said to be ECA rules. In this case, in addition to the external events and user defined or application generated queries or updates, the database can change its state on some triggers from the actions performed by the database itself. The following payoffs can be visualized from this approach:

1. Applications already existing for passive DBMSs need not be changed.

2. The DBMS can optimize rules similar to queries.

3. The functionality of the existing DBMS is enhanced.

4. Such an approach can also meet the time-constrained requirements of applications.

There are several approaches for supporting different data types. They can be formulated into three broad categories. The first is the hardwiring of the system. Here the different data types are hardwired into the system and cannot be changed. They are thus rigid. Geographical Information Systems form this kind of system. Another way is to build an extensible transactional store. The term transactional store is used to specify that the persistent storage takes care about the concurrency control and recovery management aspects. The operation to store forms a transaction and all the above mentioned properties apply. In this case the user can extend the data types required for storage but they form part of the same transactional store. The additions are made to the storage manager. The extended data types could be like R-trees, other ADTS, etc.

Another approach could be to have a transparent interface to several transactional stores. These transactional stores would be linked by an interface without the actual knowledge to the user about the kind of storage being used. This approach is the one used by us to be incorporated into the Sentinel project. In this way we can support heterogeneous data types like images, software, files etc. It is the system responsibility to make sure which kind of data goes into a particular storage.

Let us first discuss the initial objective of incorporating active capabilities and the execution model required. To efficiently incorporate the ECA rules into the DBMS, nested transactions seem to be an ideal model. Some basic concepts about the execution model and nested transactions in particular are introduced.

In the current multi-user environment today, proper coordination is required among the data operations by these users to prevent incorrect results and to preserve the consistency of the database. Various concurrency control methods have been developed over a period of time. A transaction is defined as the unit of concurrency control as in [8]. Each of these transactions has the four basic properties of ACID, defined in [11]. These properties are defined as Atomicity, Consistency, Isolated execution and Durability.

In most current DBMSs the transaction manager is designed for using a single level control structure. A standard solution of two-phase locking has been accepted as the most common and widely used method to achieve concurrency [1, 10]. In most conventional DBMSs, which are usually relational, the amount of data stored is usually very large and the type of operations involve not only simple queries, but very complex transaction operations involving a number of joins and sort operations. As can be easily seen, the conventional single level transactions do not obtain the optimal flexibility and performance required. Especially in distributed systems, a more general control structure supporting reliable and distributed computing more effectively, is highly desirable and efficient. Finer grained concurrency control and recovery and more decomposition of transaction operations is a high concern area as envisaged by the concept of nested transactions proposed by [16]. Large transaction routines are decomposed into smaller arbitrary transaction routines having a nested or inner control structure. Easier composition of transaction routines is allowed by running each of these compositions as a transaction in its own right, and also providing concurrency control within the transaction to achieve consistency and congruity.

Two levels of transactions are obtained: top-level transactions and nested or subtransactions which are separate pieces but compose to make a top-level transaction.

From these two levels we can dynamically decompose a transaction into a hierarchy of subtransactions, in effect preserving all properties of a transaction as a unit, and also assuring atomicity and isolated execution for each of these subtransactions, spawned from the top-level or the parent transaction. This results in a highly complex structure with nested worlds of synchronization.

Consequently, these subtransactions play a useful role in a distributed system where they are usually distributed among the various processor nodes to perform subtasks for the top-level transaction. Intra-transaction concurrency, which is a necessity for optimization in distributed systems, does not change the meaning of transactions but offers improved performance. This is not the strongest advantage of nested transactions as specified in [12]. The semantics of the transaction model is significantly extended by the finer grained recovery possible with nested transaction.

1.1 Advantages of the Nested Transaction Concept

Proper formalization of these pros and cons of the concept of nested transactions is required. Most of the uses pertain to a distributing computing system, but the advantages for non-distributed systems cannot be overlooked. Let us briefly discuss the advantages in a conventional computing system :

- Intra-transaction Recovery Control:

  In a conventional single level transaction management, UNDO recovery brings us back to the *begin_transaction* state. This in-transaction UNDO is considerably refined with the introduction of the concept of nested transactions, where without affecting the parent or top-level transaction or any other subtransaction, an uncommited subtransaction can be aborted and rolled back. We thus have a more graceful response to failures.

- Intra-transaction Parallelism:

  If, in an appropriate and efficient way, suitable granules for concurrency control and accessing methods are provided, then the nested transaction concept presents a reliable control structure to support safe intra-transaction concurrency. This increases the efficiency while decreasing the response time. If the transaction is large, then the potential for concurrent execution of its subtasks is also enhanced. Modularity is thus enhanced by these transaction routines (subtransactions) executing concurrently.

- Explicit Control Structure:

  The reliability of transaction processing is greatly increased by providing an explicit control structure which allows for the delegation of work, and more importantly, their atomic execution.

- System Modularity:

  Design and implementation of independent transaction modules, which are subtransactions, facilitates the safe composition of transaction programs. Other design goals viz. security, encapsulation (information hiding), failure limitation are also achieved by this system modularity.

- Distribution of Implementation:

  Robustness of any particular system may be improved in various ways depending on how the distribution is done. Implementation of distributed algorithms embodied by a flexible control structure for concurrent execution is supported by the nested transaction concept. Overall efficiency is thus increased by the distribution of data and also by distributing the processing among various nodes.

- Flexible use of Implementation Techniques:

  Types of processing or the various hardware environments offered are not an impediment to the concept of nested transactions for implementation optimization. Although a uniform and homogeneous implementation of a system implies a simple system structure and minimizes some costs, it is usually not satisfying from the performance point of view.

It is easily seen that the advantages of the nested transaction concept are not fully brought out in a centralized DBMS, their potential is exploited more in a distributed system. The concept of open nested transaction was introduced for centralized systems [9]. A special kind of open nested transactions called multi-level transactions were proposed for the multi-level structure and concurrency control for centralized DBMS operations.

1.2 Execution Models for Active Databases

Active databases are another useful area where the concept of nested transactions greatly simplifies the semantics and structure of the execution model. In the HiPAC model, the execution of the ECA rules is flexible also allowing for their decoupling. Various coupling modes are defined for a model based on nested transactions. Apart from executing rules in a conventional user transaction, the HiPAC rules can be executed in other ways. The ECA rules consist of the triggering Event, a Condition to be evaluated, and an Action to be executed if the evaluated condition is true.

The active database concept allows for greater flexibility and modularity if the rules are treated as separate entities from the transaction. These have been modeled based on the fact that every rule triggered can be constituted as a separate transaction. Different application and model semantics govern the proper formalization of these rules. The grouping of detection of the Event, evaluation of the Condition

and execution of the Action, depends on the model. These are described in detail in Chapter 3.

The remainder of the thesis is organized as follows. In Chapter 3 we state the problems encountered in active databases. A description of the various modes of execution of the rules is given. The work is extended to include the information required and the logical tables needed. Chapter 4 gives an overview of the O-O concept of the databases. Later we also give a description of the prototype acquired from Texas Instruments Inc. which was the basis of our work. In Chapter 5, the interfacing work done to link Zeitgeist with a Persistent Storage Manager (Ingres in our case) is given, apart from the brief overview of how the objects are translated and stored into the persistent storage, and later retrieved.

CHAPTER 2
PROBLEM STATEMENT AND OUR APPROACH

2.1 Motivation

On the triggering of an ECA (Event, Condition, Action) rule, the processing of that rule is not definite in terms of the time when it should be done. Depending on the application and semantics, the rule processing is done. In many cases, the normal flow of control or the execution of the program need not be interrupted on a rule being fired. The survey shows that multiple rules, triggered by the same triggering condition are usually evaluated in a sequential order, without much flexibility on the execution of the operations left in the triggering transaction.

Some conflict resolution strategy is applied to obtain the order in which the rules can be processed. We specifically avoid using the term *execution* of rules to clearly identify that the rule has three basic components. Only the Action of the rule is executed whereas the Event and the Condition are evaluated. With the proper implementation of Nested Transactions the triggered rules can be executed concurrently, apart from the uninterrupted flow of the triggering transaction segment. Also, the three components of the rules can be separated as in some models.

Another motivating factor was the goal of providing flexibility to the end-user or application programmer to spawn subtransactions and specify the segments for concurrent operations. This would help in some cases of optimization being provided by the user. Problems encountered in Distributed Databases was another factor affecting

our research. Here, one transaction accessing data at multiple sites can spawn subtransactions for each of these sites. The overall structure of such an implementation is much cleaner and modular than most currently existing ones.

Motivation for the second part of the thesis was provided by the need to incorporate the multiple data representation concept in the current existing system. Data representation in almost all of the current systems is hardwired and rigid. The basic data types are common for nearly all the systems. For more efficient data representation more forms of basic data types have to provided. Genesys at UT Austin does provide such modules but flexibility is non-existent. The building block approach used there is different from our extendible approach. We wanted the interface to be transparent to the user. Also the addition of more data types was seen as the key factor. If more data types are required then another can simply be added with the same interface to the system. The object translation need not be modified, but some form of communication should be present to keep the system informed about the addition of another data type.

2.2 Requirements

In order to accomplish the goals of the Active O-O paradigm we need to have a separate Transaction Manager for O-O DBMSs. The Transaction Manager has to be sufficiently flexible to allow for the subtransactions to exist in their own right. The subtransactions would mostly be independent atomic units, similar to a transaction, but have to *report* to the parent transaction upon completion. By reporting, we mean that each subtransaction has to be answerable only to its parent transaction. All system data or information at commit time is given to the parent transaction. Rugged concurrency control mechanisms like two-phase locking need to be provided for proper coordination among the subtransactions and the parent-child transactions.

The recovery control should be clean enough to recognize the abort and rollback of a subtransaction versus that of a top-level transaction, in which case all the operations or effects of the child transactions have to be undone.

We intend to test the open OO DB architecture to support more than one transactional store. Our prototype, Zeitgeist, conforms to the open OO DB architecture and is made up of extensible modules. This made a perfect platform for the implementation of our idea. Changes to only the interface would be required to test the design. To achieve such a platform we first require Persistent Object Store.

2.3 Approach

The long-term goal of this project is to have an extended nested Transaction Manager to fulfill the needs of Active Object Oriented databases, and to provide capability for multiple data representation.

Working towards our long term goal we first required a persistent Object Oriented database capable of operating in a multi-user environment. A prototype O-O database was acquired and it was made functional. This database called Zeitgeist used Oracle as its persistent storage. This storage manager used at the original site in Dallas was not available to us. Therefore, our approach was diverted into two distinct fragments.

Firstly, modularity, as portrayed in Zeitgeist, was kept intact by adding a Persistent Object Store (POS) to the existing O-O DBMS. This involved writing an interface to store the translated objects as obtained from the Object Manager to the underlying persistent storage. These object, as they are marked down for persistence, are transferred to the interface which takes care of the storage, and later, retrieval aspects. This essentially makes the O-O DBMS a multilevel transaction system with two distinct levels of abstractions. The first being Transaction Manager of the top

layer, and the second is the Transaction Manager of the underlying POS. Commercial Ingres, available at the Database Research Center, was chosen as the POS. After making the Persistent Storage operational changes to the interface of object manager and the object translator have to be made to have multiple object stores.

The second part was the design of the nested transaction model suitable for the existing O-O DBMS. A lot of theoretical work has been done on nested transactions as such, but the implementations aspects are not easily available. Also, there are not many systems which support such a transaction manager. The basic design given by [16] was selected and extensions were made to it to suit our needs. The concurrency control issues and the structure of the transaction tree as addressed in [12] were modified accordingly to conform for implementation in Zeitgeist.

CHAPTER 3
RELATED WORK

Apart from the issues mentioned in the previous chapter, the concept of subtransactions finds a useful part in the active database concept. The rules can be considered analogous to subtransactions. In this chapter we explain some previous work done in the field of rule generation and processing and then relate it directly as to how they can be directly incorporated into the nested transaction concept. The execution model of various approaches have been discussed and the HiPAC approach, which seems best suitable for our needs is employed later in the design aspect.

Primitive active capabilities have been incorporated into databases quite early. Triggers have been defined quite explicitly in databases like System R [7, 6] but the functionality of most of these systems have been limited to the enforcement of integrity constraints. Deductive databases has been another area where the rules have been commercially incorporated and supported. These, however, cannot be considered active as they respond only to explicit queries from outside world and hence are passive in the true sense.

There have been other active knowledge representation and inferencing mechanisms for use in AI systems which have been supported by production rules, daemons and active objects [19, 20, 15, 18]. A major drawback with these kind of expert systems is that they are inherently main memory based and thus the capability to handle rules is small. Only single threaded execution of rules is allowed which restricts such rules to be incorporated only in small databases.

Current work now tends to integrate rules at the system level, having the rules stored in the secondary memory. Some of current commercial databases include Sybase [4], Interbase [13] , Ingres (Version 6.0+), and Rdb/VMS. Some restrictions with varying degree are present in these systems. Database events like *insert, delete* and *update* can trigger rules, but event part involves operation on only one relation. None of these systems support the specification of rules on temporary or derived data. Also, the mode of processing and execution of rules can only be in a particular mode. Various other possibilities (modes) are described later in the chapter.

As is evident that nested transactions are also very well suited for active databases, we further describe in more detail the availability of different semantics in these various systems. As more and more flexibility is provided to the rules, they tend towards separate atomic operation blocks. Later, we see how these operation blocks are analogous to transactions except for a few less rigid properties. These in turn are then modeled into subtransactions. A brief description of the Starburst model (IBM Almaden Research Center [23, 22]) is given followed by the approach taken by the HiPAC [2, 3, 5] research group at Xerox Advanced Information Technology.

3.1 Starburst Approach

Starburst incorporates a production rules facility in a relational database system. Production rules are of the form *when* X *then* Y, where X is the triggering condition and Y is the action. These production rules are also set-oriented conforming to the current trend of set-oriented approach of relational data manipulation languages. In other words, these rules are also triggered by sets of changes to the database and also may eventually perform sets of changes.

Data manipulation operations are grouped into operation blocks, and these operation blocks always finish and execute indivisibly. Therefore, each user transaction

is called an externally generated operation block. Explicit specification, on the triggering of a rule, is given regarding the time they should be processed. When a rule is triggered it is added to the a logical table. Each rule is to be processed only when the last operation in the externally generated operation block has finished.

In the event of multiple rules being triggered, they are all stored in a logical transition table. After the last operation of the externally generated operation block i.e. the triggering transaction, these rules stored in the logical table are executed. One rule is selected from the set of rules that have been triggered and is executed. First, the condition is evaluated and if the condition is found to be true, then the action is executed. This is deferred mode, where the evaluation of the condition and execution of the action are delayed until the end of the triggering transaction, but before any other operation in a subsequent transaction. This in turn, might create another set of rules, all of which are appended to the logical table, and the next operation block is considered after all the rules in the logical table have been processed.

The triggered rules go into a set called the conflict set. Usually this set could contain more than one triggered rule. Since we can execute only one rule at a time, the problem of which rule to select arises. Starburst deals this with an open hand. A few rule selection strategies have been considered and the conflict resolution can be done in many ways. Some alternatives and the prevalent conflict resolution plans are described at the end of this chapter.

This work also addresses some flexibility issues in its production rules concept. The need for execution of rules other than at the end of the triggering transaction is recognized. These extensions are emphasized more in the HiPAC approach as explained below.

3.2 HiPAC Approach

In most of the prevalent systems the *coupled* mode of execution is defined. When the triggered action forms a part of the triggering transaction, and the atomicity requirements are applied to the combined execution, we call it the *coupled* execution. Either the rule execution takes place in the immediate mode, i.e. as soon as the rule is triggered, the condition is evaluated and the action is executed, if the condition is true, or the evaluation of the condition and rule execution takes place at the end of triggering transaction. A strategy for coupled execution can be formulated which specifies the triggered actions to be executed as a part of the *triggering* transaction. Whether the rules are executed immediately or delayed until the end of the triggering transaction, is an implementation and application dependent issue.

The HiPAC model allows for the *decoupling* of the triggering event, condition evaluation and the execution of the action. There are many advantages of breaking off condition evaluation and the actions into separate threads of execution. In some applications the need for actions to rules need not be immediate and could be unrelated to the triggering transaction's operations. In such cases the rules can be executed separately. An obvious advantage is the availability of system resources sooner and hence the improvement in transaction response time.

In the decoupled execution strategy, flexibility is provided to the user to specify when the condition needs to be evaluated or the action is to be executed. This directly enhances the flexibility of scheduling of transactions. This decoupled execution strategy is incorporated for active DBMSs, where the ECA rules are broken down into other separate transactions, where each of these rules can either be in a single transaction or a separate transaction is spawned for the rules. The precise execution semantics for such a system are described later.

The coupling modes are as under:

1. Evaluate condition C and execute action A immediately when event O occurs, but before the next operation in transaction T.
2. Evaluate C immediately when event O occurs and execute A after the last operation in T but before T commits.
3. Evaluate C and execute A after the last operation in T, but before T commits.
4. Evaluate C immediately when event O occurs and execute A in a separate transaction T'.
5. Evaluate C after the last operation in T and before T commits, but execute A in a separate transaction T'.
6. Evaluate C and execute A together in a separate transaction T'.
7. Evaluate C in a separate transaction T' and execute A in another different transaction T''.

The HiPAC model specifies a special *cause-and-effect* relationship between the triggering transaction and the triggered transaction. The following *causality* criteria should be satisfied in addition to the transactions being serializable.

- If transaction T causes transaction T', then T must be serialized before T'.
- If transaction T causes transaction T', then T' can commit only if T commits.

3.3 Extension to Starburst Model

The following work was done structured on the Starburst model. There is one main distinction. The original Starburst works allows only for the net case, i.e. all

the externally generated operation blocks are indivisible. For the immediate case, we have to allow for the occurrence of the non-net case also. Here the execution of the externally generated operation block can be stopped for some time while the condition (and action, if necessary) are attended to. A more generalized model is considered, where such an restriction is not present.

3.3.1 Immediate(C) - Immediate(A) mode

The immediate mode is not very difficult to handle. In the deferred mode, the triggered rules are put in a conflict set, as they are triggered, and when the user generated operation block ends, one of these rules is chosen by some selection strategy, and the condition is checked. The action is subsequently carried out after that. While employing the immediate mode, we would still need to have a conflict set, in case more than one rule is triggered by any statement in the user generated operation block. But instead of waiting for the operation block to end, the condition for one of the rules is checked at that instant and, if true, the action is carried out.

Each transaction can be defined as comprising of several transitions, the first transition is always an user generated operation block. If no rules are triggered, then this is the last and only transition in the transaction. If at some instance in the externally generated operation block, a rule is triggered, then further execution of the operation block stops. Another transition begins, with the condition of the triggered rule being checked and the action being subsequently executed. This continues till the end of the operation block and till the *conflict set* is empty. The transaction is said to end at this point. The semantics for the logical tables is similar as the condition and action are executed at that instant and not at a later time. The net effect of two transitions, which was combined in the deferred mode, has to be taken

into account individually. All insertions and deletions have to be recorded, as the condition is checked immediately.

3.3.2 Immediate(C) - Deferred(A) mode

This case is trivial and is similar to the above in most part. But we need to have the logical tables here, as although the condition is checked immediately, the action is carried out at the end of the transaction. This is necessary as the action could also be on the effected tuples. This is a combination of the above and the deferred mode.

3.3.3 Deferred(C) - Decoupled(A) mode

Another interesting case is where the condition is evaluated in the deferred mode but the action takes place in the decoupled mode. For the transaction triggering the rule or rules, the same semantics of the Starburst model are applied for the condition evaluation. As soon as the condition is evaluated and before the commit of the transaction, the action part of the rule selected from the conflict set, which is in another transaction, can be executed. Now the two transactions can be run concurrently. We can have more than two concurrent transactions from this structure based on the number of rules that were triggered.

One distinction has to be cleared out. Each transaction in this mode consists of transitions. The first transition could be an externally generated operation block and then followed by one or more transitions. Each of these transitions is composed of the condition evaluation statement of the rule selected from the conflict set. The action part of the rule has to be carried out in another transaction. Apart from the condition evaluation statement the transition could also have another statement which creates another transaction where the action is carried out and this transaction runs concurrently. This new transaction is similar in structure to the initiating transaction except for the first transition. The initiating transaction continues till

all the rules in its conflict set have been accounted for, i.e., their condition has been checked and the appropriate action taken.

Usually, at the end of the transaction triggering the rule/rules, the logical transition tables are deleted. This cannot be done in this mode, as the new transaction which was created for the action could still be running and might require some information about the affected tuples. Therefore, these logical transition tables need to be deleted only when all the newly created transactions, for the rules, have committed/aborted.

3.3.4 Immediate(C) - Decoupled(A) mode

The mode is similar to the one described above. The initiating transaction is further broken down into more transitions which comprise of the condition of the triggered rule being checked. The logical transition tables being maintained here would be semantically different from those maintained in the Immediate(C)-Immediate(A) mode. As a very simple example, we cannot combine the effect of two transitions while making a transition table. Each tuple affected by *insert, delete or update* has to be inserted.

3.3.5 Decoupled(C) - Decoupled(A) mode

Here each transaction consists of either the user generated operation block or the *condition-action* part of the rule triggered by the triggering transaction. This triggering transaction could also be internally generated. One such transaction is created for each rule in the conflict set. Two variations of this mode are possible, but they are semantically same. Either, the condition is checked and the action is carried out in a single transaction, separate from the rule triggering transaction, or the condition and action are also separated. Either implementation will not alter

the semantics much. The logical transition tables would be deleted only after the completion of all the transactions with the action part of the rule.

The semantics for the maintenance of the logical tables is as follows. These logical tables contain tuples. The *inserted t* table is kept updated with the execution of each *when inserted* rule. Similar concept can be applied to the *deleted t* and *updated t* tables. Those tuples which have been modified are kept in these tables. As the execution proceeds, more affected tuples are added to these tables. An important point to note here is that we assume the concurrent execution of these different transactions comprising of the *condition-action* part of the rule. Therefore, we need some kind of a locking mechanism so that two such transactions do not access the same table at the same time. Implementation of this semantics will not be difficult. Concurrency control methods like standard two-phase locking, timestamping can be applied.

3.4 Conflict Resolution

Conflict resolution, as the name implies, deals with the selection of a particular rule to be executed from a set of rules triggered by a transaction. What selection strategy should be applied to execute a rule ? This is to make sense or explain why a particular rule was selected and not a different one. Various alternatives are available. Let us look at them.

1. Select or chose a rule at random (arbitrarily) from the conflict set. This implies that all the rules have an equivalent probability of execution. As such, this seems a very non-deterministic way of rule selection. This is undesirable in most systems, where we would want some kind of ordering and control over the rule selection procedure.

2. Select a rule at random from the set of rules that was activated by the first operation. There could be just one rule in such a set.

3. We can assign priorities to rules when they are triggered, and later select the rule based on its priority. This implies total ordering of rules. The rules are in a total order and hence guarantee us a deterministic behavior. Since we want to define rules independently this might not be possible.

4. The rules could also be partially ordered and one is selected such that no other triggered rule is strictly higher in the ordering.

5. A specific weightage can be given to rules such that a *weighted ordering* is implied. This can be something like choosing a rule based on its ordering and also the age of the triggering transaction or the rule itself.

Independent rule definition is again precluded if the rule selection procedure is defined at the time of the rule definition. But this kind of limitation is acceptable if at any given time all the rules have the same selection procedure. These procedures, as explained earlier, could be FIFO, random, totally ordered, partially ordered, weighted ordering etc.

The best alternative seems to be the weighted ordering of rules, in which most of the factors affecting the rules are taken into account before selecting a rule. Consider a case where some higher priority rule is triggered by an operation. The execution of that rule might trigger another high priority rule. This new rule triggers the original rule again. This kind of semantics prohibits the execution of all the other rules and we have an infinite rule execution structure. A weighted selection scheme will eventually give priority to some other set of rules which have been triggered earlier and are

waiting for a long time now. Thus the infinite rule cycle can be broken. Some other run-time detection mechanism like a timeout can be introduced to break cycles.

This conflict resolution also plays an important role when the concurrent execution of rules is envisaged. If the resources and the semantics of the system permit then the execution of these rules need not necessarily be in a serial order. The ideal solution would be to execute as many unrelated rules as possible under the environment. Concurrent execution of rules does not rule out the selection strategy. This still remains in effect, in case two sets of unrelated rules have to be executed. We would again need some conflict resolution about which group to be executed at first. This type of conflict resolution involves many more factors than the ones given above for the rule selection. In these cases many of these factors would be system and application dependent and hence are not discussed in this chapter.

CHAPTER 4
OVERVIEW OF ZEITGEIST

4.1 Object-oriented Concept

It has been observed that the capabilities of record-oriented data models are limited in capturing the complex structural relationships and the behavioral properties of objects in advanced application domains such as engineering, scientific, statistical and military applications. Many semantic models [21] and object-oriented (O-O) data models [14] have been developed to alleviate the limitations of record-oriented data models. The O-O models provide a variety of modeling constructs, which simplify the task of modeling complex data. The main features of an O-O data model are:

1. They support the unique identification of objects by system assigned object identifiers.

2. They support abstract data types and allow complex objects to be defined in the form of hierarchies.

3. They allow encapsulation of structure and behavior within objects and classes.

4. They allow definition of hierarchies and the inheritance of structural and behavioral properties among object classes in these hierarchies.

As can be seen, transactions in our system are significantly different from transactions in conventional transaction processing systems based on less expressive data models.

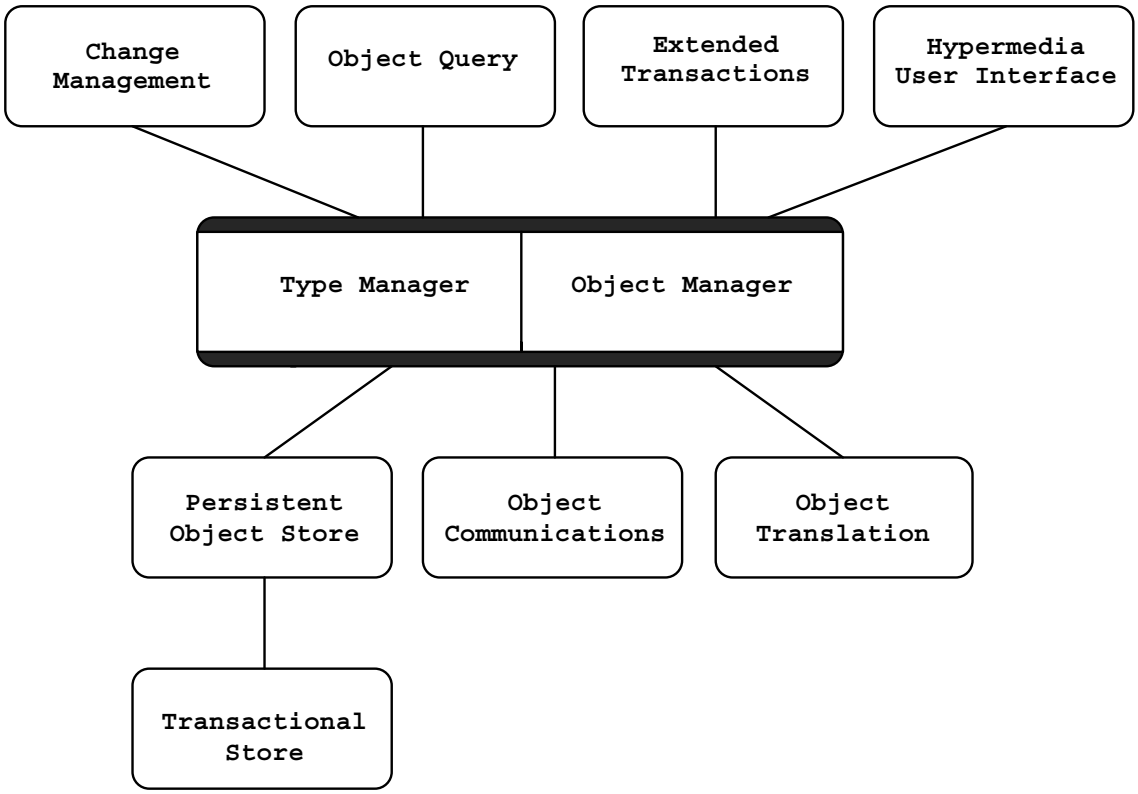

Figure 4.1. Zeitgeist Modules

We further extend our work into the Object-Oriented paradigm. In a hierarchy similar to the Object-Oriented design schema, the nested transaction model can be easily adopted. A prototype O-O model called **Zeitgeist** or *free spirit* was used extensively in our work [17]. This prototype has been acquired from Texas Instruments Inc., Dallas and was the basis for our design.

4.2 What is Zeitgeist ?

Before going further we will describe the execution model, particularly the Transaction Manager of Zeitgeist. This O-O DBMS originally used two kinds of storage structure for its persistent storage. The Open O-O database architecture, on which Zeitgeist is based, is shown in the figure 4.1.

Zeitgeist, which has been used as our platform has the following features. It has a Persistent Object Store (POS), an Object Management System (OMS), a primitive OQL, and a two-level flat Transaction Manager. Detailed discussion about the POS and the Transaction Manager is given later.

C++ been used to define the O-O conceptual schema and has also been used to manipulate the internal object structure. To make objects persistent the language has been slightly modified to P-C++ which is pre-processed to conform to our needs.

The persistent storage in the actual Zeitgeist is either the file-based structure on the Unix platform, or the Oracle relational DBMS. Various aspects are compromised if the Unix file-based storage structure is used. In this mode, all of the database objects and the related information is stored in a file. When any object is accessed, the whole database is loaded into the main memory. There is an obvious disadvantage of working space if the database is large. Another drawback comes into picture when the multi-user environment is envisaged. There is no way to ensure proper concurrency and consistency of the data.

Since nowadays, working on single user systems is out of question, it was futile to work on that and we needed a multi-user system. The other alternative was the Oracle relational DBMS. At the time of writing this, Oracle had not been installed, but commercial Ingres was found available. An interface was written to link with the Ingres database, which was used as the storage manager. The multi-level transaction model comes into picture when we look at Zeitgeist with its underlying storage manager. The relational coupling provided by the storage of translated objects from Zeitgeist is unique. A well defined interface on two levels of abstractions is available. The first is the Zeitgeist's own transaction manager, and the other level of abstraction is the transaction manager of the underlying storage, which is the Ingres DBMS

in our case. The next chapter describes in detail how the interface to Ingres was approached and its functions.

4.2.1 Data Definition

Schema definition or data modeling is done in Zeitgeist using the C++ programming language. The pre-processor for C++ has been slightly modified to recognize certain keywords using while designing the schema. These additional constructs make the definition persistent and hence the language has been termed as P-C++. The Data Definition Language (DDL) is the type definition part for application classes and information for run-time management of objects created, saved and restored by various application programs. This DDL translator imposes some restrictions on the type definition. The inheritance concept of C++ classes is used extensively to provide inheritance among hierarchical objects. The various relationships, One-to-One (1:1), One-to-Many (1:m) and Many-to-Many (m:n), to relate instances among classes are provided by two other type of relationships, embedded or referenced. Different combinations of these two can generate the required relationship. There is not a great deal of difference between these two. Usually embedded instances are used for simplicity of notation and use. Since they are static, dynamic allocation of instances and dereferencing of pointers is not possible. In cases where the size of the objects and instances is not known in advance, the referenced instances are used. This dynamic behavior is useful depending on the application. More details are available in the Zeitgeist Persistent C++ User Manual [17].

Objects or instances of classes can be either transient or persistent. The transient objects exist only during the application program's lifetime and cease to exist after that. Conventional C++ class definition is sufficient to provide this kind of behavior.

These classes need not be processed by the DDL translator. Whereas, objects designated for persistence have to be pre-processed as C++ by itself is unable to capture the notion of persistence. An object is also persistent if it is in the transitive closure of another persistent object. These persistent objects can be of two kinds: Independent and Dependent. The differences between these two comes from certain properties like designation for persistence, referencing from an object or more than one object, storage method, and method of retrieval after being stored by P-C++. The following tables summarize the differences and similarities between the Independent Persistent Objects (IPO) and Dependent Persistent Objects (DPO).

Table 4.1. Characteristics of Independent and Dependent Persistent Objects

Characteristic	Independent Persistent Object	Dependent Persistent Object
Can object be persistent?	yes	no
Can object be embedded?	yes	yes
Can object be referenced?	yes	yes
How can object be referenced?	C++ pointer '*' P-C++ PTR	C++ pointer '*'
Referenced from how many objects?	'*':exactly one object PTR:any number of objects	exactly one object
Independently accessible?	'*':no PTR:Yes	No

When the DDL class definition is given, objects which are to be made persistent are specified. The declaration is same except for the keyword `persistent` appearing as `class temp : persistent`. The class definition can have private data and function members and also protected data and function members. It can also have public function members but no public data members. This restriction of no public data members is applied to both IPOs and DPOs. At the transaction commit time,

P-C++ saves the embedded data item or object as part of the containing object. Referencing of objects can be done using the two pointer data types: C++ pointers (*) or P-C++ pointers (PTR). More information is available in the Zeitgeist User's Manual on the use of these pointers.

Demon routines, which are invoked automatically by P-C++, can be specified on certain conditions, like designation of IPOs for persistence, commit to the database, fetch from the database, or for fatal errors forcing the current process to abort. Some restrictions do however apply on these demon functions. These demons can be of private, protected or public visibility. They cannot take any arguments and have to return an integer value. Another restriction is that they have to be public function members of a persistent class. To avoid recursive invocation of demons, a demon cannot call any other P-C++ method. Currently, Zeitgeist supports the following four demons: Persist, Commit, Fetch and Abort. As the name implies they are invoked when an object is designated for persistence, when a commit is called, when an object is fetched from the database, and when a transaction aborts, respectively.

4.2.2 Application Programs

Having written the schema, the user then develops application programmes to access and manipulate data. An instance of the Zeitgeist class has to be created before any methods can be called. There are quite a few ways to create an instance of this class. If an instance already exists, another one is not created. This implies that there can be only one instance of the Zeitgeist class per process. Once the instance has been created, the user can start and stop the system (Zeitgeist) as many times as necessary. During a startup, a signal handler is established for many signals. This signal handler takes control on some prespecified signals, prints the error message on the error log and aborts the process. The signal handler accepts signals from the

operating system, the application program, or other error *signals* generated by the user. If the user also defines a signal handler, then the user's signal handler takes precedence over the built in one and executes whenever a signal occurs.

A Name Manager is also provided by Zeitgeist to accept and maintain the association of names to OIDs/IPOs. These names are stored in a context. The user can have as many contexts as desired but they have to be unique. Object names have to be unique in a context and the same name can appear in different contexts. Also, two names across different contexts can be associated with the same object. These contexts and names have to be null terminated character strings. A default name context called *default-context* is established during startup. A new name context can also be established by calling the specified method. After a fetch or creation of an object, it can be explicitly named. This implicitly designates the object as persistent, but to save it to the database it has to be indicated that it has been modified. Multiple names, in the same or different name context, can be associated with the same object. Similarly, unname of an object is also supported. This is also logical. Even after an object has been unnamed, it can still be accessed directly by the IPO.

4.2.3 Transaction Manager

Maintaining consistency is an integral part of any database. To insure this three transaction boundaries are defined in Zeitgeist: *begin_transaction, commit_transaction* and *abort_transaction*. *Begin_transaction* call starts a user transaction and assigns a unique identifier to it. This id is unique throughout the system, even if multiple users are accessing the database. Since only one instance of Zeitgeist is allowed to be created, it helps in keeping a track of the different transaction ids and avoids duplicates. The other two calls are also self-explanatory, with *commit_transaction* not only implying a commit to the Zeitgeist database, but as we shall see in the next

chapter, it also sends an explicit commit to the underlying storage manager. Different versions are supported and the previous versions of objects are not discarded or overwritten. The timestamp for the previous versions is saved and they can be later explicitly retrieved. If a time context is specified then the version saved at or before the specified time is retrieved and installed in the main memory. Two phase locking protocol is employed in Zeitgeist and it is only during the commit or abort time that the locks held on various objects are released. Once a lock on any object has been released no additional locks on any object can be acquired. A physical lock on an object is acquired only when the application requests one or when the commit is processed, whichever comes first. A higher degree of concurrent access is obtained by following this policy, rather than acquiring locks at the earliest possible time. Similarly for assigning unique oids, this occurs when the object is made persistent or the current transaction is committed. However, the user has the flexibility to specify a storage group number, otherwise, the system assigns its own default storage group. The user defined storage group number must exist (see next chapter).

Zeitgeist also allows the application programmer to check the memory residency and the lock status of an object. The method is analogous to the fetch function call. Either the name of the object or the IPO class PTR can be specified. The optional time context uses the latest version if omitted. A method *time* is provided, which returns the system time (a 32 bit integer), to specify the time context for subsequent calls to *fetch*. These locks being totally ordered can be checked for and upgraded if required. The first table below gives the various lock modes.

Retrieval of objects is done with the *fetch* call. A time context and concurrency lock can be specified at the time of retrieval. If they are not given then the latest version object is fetched with a READ lock. As previously specified, the object can be either fetched via the name context or with the P-C++ PTR for the IPO class.

Table 4.2. Zeitgeist Lock Values

C++ Enumeration Name	Specifying
ZG_NO_VALID_LOCK	No lock held
ZG_READ_ONLY	Read Only lock held
ZG_READ	Read lock held
ZG_WRITE	Write lock held

As we have seen earlier that, all objects referenced from a particular persistent object (IPO) are associated together, they are installed in the main memory when this IPO is fetched. This implicit retrieval of objects increases concurrency and also reduces the number of accesses to the Storage Manager.

When an object is created, an implicit WRITE lock is granted since this is the only process which has access to that object. Apart from indicating that an object is to be made persistent, all newly-created objects have to also be marked as modified for them to be saved to the database. This indication of *modified* also holds good for all retrieved objects which have been updated. If they are not explicitly marked as modified the new version will not be stored in the database. This method *set_modified* prompts for attempts to obtain a WRITE lock on the modified object.

This table gives a compatibility matrix for lock upgrades. This shows that *upgrade_lock* can also be called even if no lock is held on the object.

While retrieving an object, the complete set of versions of the object is locked. Sometimes it is necessary to retrieve an object without blocking the other processes to read or create new versions of the same object. To facilitate this, a third lock mode called READ_ONLY is supported. In this mode, the application program retrieves a specific version of an object without locking the complete set. A time context, other than most recent, has to be specified in this mode. Guaranteeing hardware and software tolerance, i.e. no failures occur, this fetch will always succeed. This implies

Table 4.3. Compatibility of Lock Upgrade

Current Lock	Upgrade to	Result
None	ZG_READ_ONLY	ZG_READ_ONLY
None	ZG_READ	ZG_READ
None	ZG_WRITE	ZG_WRITE
ZG_READ_ONLY	None	Error (-1)
ZG_READ	ZG_WRITE	ZG_WRITE
ZG_WRITE	ZG_READ	ZG_WRITE
ZG_WRITE	ZG_WRITE	ZG_WRITE

that, this lock mode cannot be upgraded to READ or WRITE, and the object has to be refetched with the appropriate lock mode. The lock mode and time context is also propagated while retrieving an object. If an object is referenced by two other object retrieved with different lock modes, then at the time of retrieval of this object the stronger of the lock modes is used. The lock modes are in a total order: READ_ONLY < READ < WRITE. Similarly, the time context of the referencing PTR is used to fetch an object.

4.2.4 Debugging

Extensive debugging is provided in Zeitgeist. Environment variables can be set in the user's shell to see the details. There are two variables: ZGERROR and ZGDEBUG, where the first is the pathname and the second a symbol, point to where the error file is to be written and what kind or level of debugging is desired, respectively. The default value for ZGERROR is the standard out and the information is dumped on the screen. It is advisable to have these environment variable set in the `.login` or `.cshrc` files. The symbol can have one or more of the following values shown in table 4.4

Table 4.4. ZGDEBUG Symbols

Symbol	Resulting Output
OMS	Object Management System trace
OTS	Object Translation Service trace
OMAP	Object Map System trace
HIIO_GET	High level I/O get trace
HIIO_PUT	High level I/O put trace
HIIO_REST	High level I/O trace of misc. ops
LOWIO_GET	Low level I/O get trace
LOWIO_PUT	Low level I/O put trace
LOWIO_REST	Low level I/O trace of misc. ops
IO_ERROR	errors from all routines
IO_DETAIL	additional detail for HIIO and LOWIO
ALL	all of the above

CHAPTER 5
EXTENSIONS TO ZEITGEIST

The extensions we propose for Zeitgeist encompass the much broader framework for active databases. This thesis represents the work done in two directions towards achieving our goal. The following extensions are proposed:

1. to support rules in DDL.
2. to support an execution model for rules.
3. to support Multiple Transactional object representation.
4. to execute rules efficiently.
5. to support complex events.

2 and 3 represent the work done in this thesis. This chapter describes an execution model to incorporate rules in Zeitgeist. The various details of maintaining pertinent information are discussed. Suitable modifications for the lock table and the transactional structure are given which take us closer to our goal of making Zeitgeist active. The next chapter describes how another persistent object storage was added. The object representation in different storage managers is highlighted.

As is evident that nested transactions are also very well suited for active databases, we further describe in more detail how the different semantics are available from these.

5.1 Concurrency Control in Zeitgeist

Zeitgeist, our prototype system, can operate in a multi-user environment. The transaction information and the lock table are kept in the global space. Each transaction structure contains information identifying that transaction and has two pointers to the lock table. The first one gives us the locks held by the current transaction. The second pointer points to the locks held by all other transactions. A hash table, kept globally, is used to identify objects and hashes on the object number and storage group number, whenever a lock is required on a particular object. Each object is identified by an object number and a storage group number. The hash table contains entries of the lock table, and two pointers, one to link nodes hashed to the same bucket and the other to link nodes of the same transactions, respectively. A lock request on a particular object can be of two types. The first case is the *no-wait* type of request. Here, if one transaction is holding an exclusive lock and another one requests it, then the requesting transaction does not wait till the lock is granted. Instead, an unsuccessful attempt is propagated to the caller. It is upto the calling function to decide what to do in such a case, whether to continue to the next operation, or retry or abort the current transaction.

In the *wait* type of request, the requesting transaction waits until the lock is released by the holding transaction and then retries acquiring that lock. Deadlock detection comes into picture in this case. If the transaction is still waiting after the lock has been released, a deadlock is in effect, and a victim to abort, has to be chosen by the detection algorithm. A time interval, in seconds, can be specified while starting the **zgt_init** process. This process waits until it checks for locks held by processes which no longer execute. The frequency to check for and resolve deadlocks is also obtained from this number.

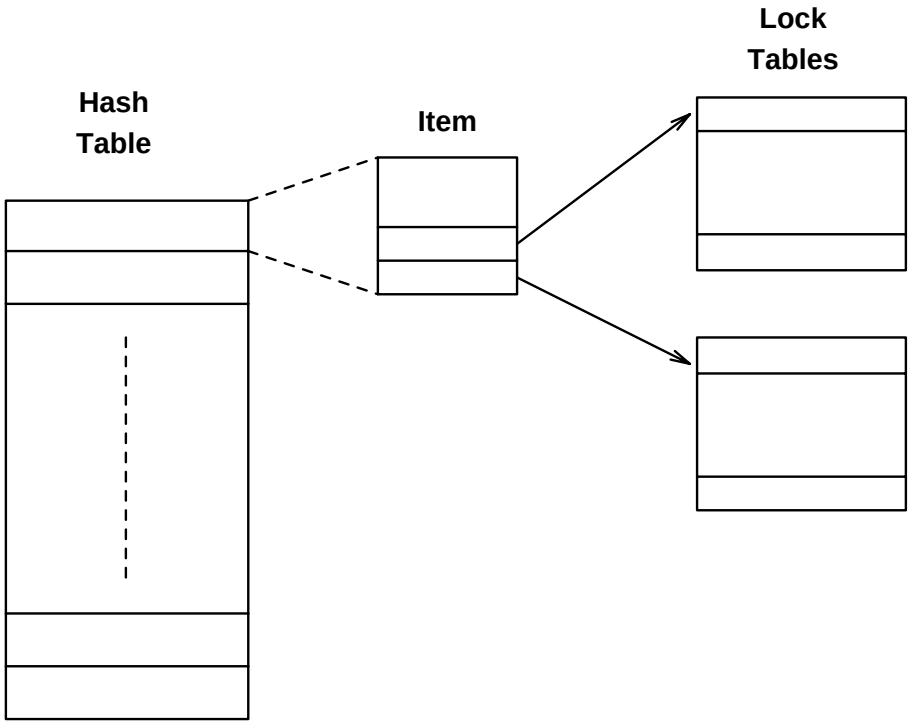

Figure 5.1. Hash Table

If a lock is requested on a particular object, it is hashed on the object number and the storage group number. If in the linked list returned, no entry is found, then a lock does not exist on that particular object, and the lock is then granted for that transaction. In case there is an entry for that lock, then it is checked if the requesting transaction owns it. We return successfully if this is the case. Otherwise, we know that some other transaction holds the lock, and we can acquire it only if it is in the shared mode. The lock is granted, if in the shared mode, else depending on the type of lock request (*wait* or *no-wait*), we return unsuccessfully or wait in a queue. If the request calls for a wait until the lock is granted, the transaction is put in a waiting queue, which is then later checked for deadlocks. In case of deadlocks, the youngest transaction is aborted. This is done on the assumption that the youngest transaction

in the system would have done the least amount of work, and thus the overhead is minimized.

5.2 Generalized Locking Rules

We now present the locking rules for our approach. The general locking rules are as described by [16]. We further extend them to suit our needs for a nested transaction model to be made functional in Zeitgeist.

- A transaction may hold a lock in the *write* mode if all other transactions holding the lock (in any mode) are superiors of the requesting transaction.
- A transaction may hold a lock in the *read* mode if all transactions holding the lock in *write* mode are superiors of the requesting transaction.
- When a transaction aborts, all its locks, *read* and *write*, are simply discarded. If any of its superiors hold the same lock, they continue to do so, in the same mode as before the abort.
- When a transaction commits, all its locks, *read* and *write*, are inherited by the parent if any. This means the parent holds each of the locks, in the same mode as the child held them, or if the parent already held these locks before, a *union* of locks is performed. The new mode is computed as such:

  Parent's new mode = MAX (parent's old mode, child's mode)

  This union of lock modes can be done since the locks are in a total order. This upward inheritance of the locks follows the multiple retainment rule while keeping the more restrictive mode [12]. This ensures that a transaction's retain or hold mode never decreases when the above rule is obeyed.

  None < Read Only < Read < Write

If a transaction retains a lock instead of holding it the following things can be visualized:

- A transaction retains a lock in the X mode:
  - no non-descendants can hold the lock in any mode.
  - descendants can hold the lock in either of the two modes.
- A transaction retains the lock in the S mode:
  - no non-descendant can hold the lock in X mode, but can potentially hold it in the S mode.
  - descendants can hold the lock in either mode.

Looking closely, we see that if a transaction retains a lock in the X mode on a particular object then any other transaction retaining a lock on the same object has to be superior or inferior of that transaction. In other words, the other transactions retaining the locks have to be in the same hierarchical path as the current one.

The inheritance of *write* locks when a transaction commits has two major effects: it allows subtransactions within the parent's microcosm (viz. all the inferiors of the parent) to see any changes made by the subtransactions and make further updates; and preventing transactions or subtransactions outside the sphere of control to make any changes to the object. The *sphere of control* of a transaction includes all and only its inferior subtransactions. Similarly, the inheritance of a *read* lock on the commit of a subtransaction, prevents updates by transactions outside its parent's sphere of control. This enables to ensure that the parent's view is presented with a consistent snapshot of the object being manipulated.

Only upward inheritance is described, as the concept of downward inheritance does not give an advantage to us, but only makes the whole design more complex. The main reason being that our assumption that when a child transaction is running, its parent is suspended. Hence, we have no concurrency and subsequently no conflict between parent and child transactions. Superiors of a transaction are not hurt with this kind of inheritance, since every superior sees only the committed data and not the contaminated one. We do not want a transaction to be dependent on the fate of its children. This *upward inheritance before commit* is not permitted which would allow a transaction to have access to the uncommitted and possibly corrupted data of its children. This firewall against undesirable dependencies is kept intact by having such restrictive rules. Another advantage of this approach is that concurrency control is considerably simplified, as there is no concurrency and conflict along the hierarchical path. This *sibling parallelism* concept is applied to our design. Therefore, as mentioned above in the locking rules, a transaction holding or retaining a lock can make it available under its sphere of control.

5.3 Incorporating Nested Transactions

We now propose to incorporate the following changes to the already existing Transaction Manager in Zeitgeist. The basic concept presented above has been followed with some minor changes to accommodate the existing structure and make the implementation more robust and efficient.

5.3.1 Creating a Top-level Transaction.

No changes are required to a create a top-level transaction. The structure of the top-level transactions remains the same. The usual data structure is used to keep information about the transaction and its lock tables. This creation of the top-level transaction follows the procedure as implemented in Zeitgeist. Our model uses the

same top-level transactions that are created and builds subtransactions under them. The data structure currently in use, with only a few modifications (to be described later), contains sufficient information to build subtransactions.

5.3.2 Creating a Sub-transaction.

The data structure has to be slightly modified to accommodate this concept. Apart from a pointer to the parent transaction, this modified version also consists of an added field in the lock table. This is to keep the current status of a lock. This new field is called *holdmode* to distinguish it from the *lockmode* already present. The *lockmode* specifies whether the lock is being held in the shared or the exclusive mode. *Holdmode* can contain the following four values:

retain(x) | —

hold(x) | —

hold(x) | *holdsub(x)*

— | *holdsub(x)*

Let us define the above terms . These are the extensions made to the nested transaction model. The terms define the status of a lock and its availability to other transactions. The algorithm presented later, uses these to find if a lock can be granted or not. The argument given in the values represents the object in question and not the mode of the lock.

1. *Retain(x)* | —

   This implies that some sub-transaction in the system acquired a lock on object x and has now released it. There is only one occurrence of a *retain* on an object in a transaction tree. If the lock was held in a shared mode, then of course there can be more than one. Since, on commit the information is propagated upwards, this means that one of the inferiors of the current transaction acquired

a lock and then released, implying that the original transaction no longer exists. If a child now acquires the lock then the value gets modified to — | *holdsub(x)*. A point to be noted here in case of retention in the exclusive mode is that access of the object is limited to only the subtransactions in the sphere of control of this transaction. Although presently no transaction is holding the lock, any arbitrary transaction in the system is not allowed to acquire that lock.

2. *Hold(x)* | —

   When a transaction first acquires a lock, this entry is made in its lock table. This states that the current transaction holds the lock and no other subtransaction holds it. If a subtransaction is spawned then this lock can be passed on to it. That child can in turn pass the lock to its children. Since, on the existence of a subtransaction, this transaction will not be running, the inheritance of locks is not dangerous and preserves the consistency of the data. However, when the lock is granted to its children this value changes to *hold(x)* | *holdsub(x)*.

3. *Hold(x)* | *holdsub(x)*

   The current transaction first acquired the lock and one of its inferiors has now inherited it. If it an exclusive lock then no other transaction or subtransaction can have this. On the commit of all of its children, the value gets changed to its original one which is *hold(x)* | — .

4. — | *holdsub(x)*

   This implies that one of the inferiors of the current transaction presently holds a lock. In a similar argument, if the lock is in exclusive mode, no other transaction or subtransaction can acquire it. Also, analogous to the above case, on the commit of all of its children, this changes to *retain(x)*.

5.3.3 Acquiring and Releasing Locks.

- Acquire a lock at the top-level.

  No changes or further search is required while acquiring a lock at the top-level. The algorithm used for conventional model suits just as well.

- Release a lock at the top-level.

  This remains similar to the current design and the entry is simply removed from the lock table.

- Upgrade a lock at the top-level.

  Here we intend to incorporate some changes to make the current design more cleaner. A compatibility matrix is presented and the upgrade of locks is performed accordingly if permissible.

- Acquire a lock in the sub-transaction.

  Two things need to be done here while checking for a lock on a particular object.

  1. check whether the current transaction is holding that lock.
  2. move to the parent's lock table and search if that transaction holds the lock. If the lock is still not found, then we need to see if any other transaction in the system is holding that lock.

  The following algorithm is followed to acquire a lock. The process is continued till the *root* (top-level transaction for this tree) is visited or one of the following from the *holdmode* is encountered. The search for locks in the lock table for other transactions is followed in exactly the same way as in the current implementation. Since information about all locks held by a transaction tree are kept at the root or top-level transaction, searching the whole tree is redundant.

```
acquire(tr_node) {
    Search in the lock table of current transaction.
    If an entry is found ; current transaction owns the lock.
    switch (holdmode) {
        case retain() :
            change to — | holdsub() ; lock available.
            break ;
        case hold() | — :
            change to hold() | holdsub() ; lock available.
            break ;
        case hold() | holdsub() :
        case — | holdsub() :
            check lockmode ;
            acquire only if in shared mode ;
            if cannot acquire,
                wait or return error to caller.
        default :
            if top-level transaction, search the other's
            lock table (only top-level transactions) ;
            else call acquire() with parent node.
        } End of switch.
    If no entry is found, search the other's lock table of
```

top-level transactions.

}

- Release a lock in the sub-transaction.

  The *release()* algorithm works in an analogous way as the above, removing the locks and makes the required changes to the lock table as it goes to the parent transaction. The current entry in the lock table is removed. Then the same function in the called for modifications in the parent transaction's lock table. The *holdmode* is checked. Only three cases are possible. The *retain()* mode will not be present, as if a subtransaction is holding a lock, then the parent will have an entry of the form *holdsub()*. After making these changes, the loop exits. This is because changes are required only in the parent transaction. If the transaction in use is the top-level, then changes are only made for that lock table.

  release(tr_node) {

  If top-level transaction

  remove entry, if present ;

  Exit from loop.

  switch (*holdmode*) {

  case *hold()* | — :

  Remove lock entry.

  break ;

  case *hold()* | *holdsub()* :

  Change entry to *hold()* | —.

break and exit from loop.

case — | *holdsub()* **:**

Change to *retain()* **;**

break and exit from loop.

} **End of switch.**

Call *release()* **with parent node.**

}

- Upgrade a lock in the sub-transaction.

  Upgrading locks in subtransactions is similar to the concept of acquiring locks in the subtransactions. We first check to see if any other transaction is holding or retaining that lock. The search along the path to the top is done. Algorithm followed is similar to the one to acquire locks. A search is done till the highest ancestor of the current transaction and then the lock tables for other top-level transactions are searched. No changes need to be made to the lock table. While committing the subtransaction, a union of lock modes is done and the parent keeps the more restrictive mode.

These algorithms have a slight modification to the design aspects considered earlier. We keep all information about a lock at the top-level transaction. Also the information is replicated along the hierarchical path, on which the lock has been acquired at a lower level. This has been done due to two main reasons, making way for an efficient and cleaner implementation.

1. The search along the entire tree or sphere of control is avoided by keeping this information along the hierarchical path. The search space is then limited to this path, from the current transaction upwards till its top-level transaction.

2. If in case a lock had not been acquired, and thus no information is available, then we need to search the lock tables of all other transactions. By keeping a larger lock table at each of the top-level transactions, we greatly reduce the search space. While searching for the lock in the other's lock table we now need to visit only the top-level transaction to figure out if a lock is held. Obviously, this method is a lot cleaner and shorter than searching the entire tree of another top-level transaction.

CHAPTER 6
INTERFACING INGRES TO ZEITGEIST

6.1 Need to Interface

Although the file-based storage manager requires substantially less overhead, both during setup and at runtime, but it also lacks features required for production use. Table 6.1 lists the database features missing when using the File-Based Storage Manager.

Table 6.1. Drawbacks of the File-based Storage Manager

Missing Feature	Implication
Concurrency control & locking	No concurrent users.
Recovery	An error during a commit may corrupt the database.
Consistent performance	Performance may degrade as a function of database size.
External storage	The entire database is cached in main memory, limiting the database's maximum size.
High cost	Very inexpensive to implement and use.

Zeitgeist is inherently an object-oriented front-end to a persistent Storage Manager. The user sees an object-oriented structure. Starting from schema definition to the query aspect, this structure is presented. Conforming to the Open Architecture design paradigm, it is made of extensible modules. Here we will discuss the Storage Manager. Each object is translated into a set of values which are then stored. The schema definition is in the object-oriented form. The available system from Texas

Instruments Inc. had two types of storage structures, as mentioned earlier. Since Oracle was not available to us, a need for having another Storage Manager with its own Transaction Manager was visualized. Interfacing was done to the Ingres relational database. The Transaction Manager of Zeitgeist works on top of the TM of Ingres. The objects made persistent in Zeitgeist are stored in Ingres in a different translated form. This data is transparent to the user. Only the top level TM and structure is available to him. Concurrency control and recovery is handled by this underlying Storage Manager. Although the basic structure of Zeitgeist supports concurrency, the storage should also be accessible by concurrent users. The file-based version of the system did not support this and the access to data had to be regulated. Hence, the need to interface was imperative.

The interface is very much similar to the one used for Oracle. First, a database is created in the underlying Storage Manager by scripts. Three tables *viz.* **Groups, Value** and **Refto** are created and initialized. Apart from these three tables, indexes are also created on each of these. The user can query the Storage Manager directly for debugging purposes. Otherwise, the data retrieved is of no consequence. The Groups table is independent and gives us the valid storage group numbers. This can be independently modified to add or delete storage groups. However, care should be taken while deleting a storage group, such that the referential integrity is not violated. A storage group number which exists in the other two tables cannot be deleted. Only when all objects from that storage group have been deleted, the number can be discarded. If this is not kept in mind, then all those object will not be accessible from Zeitgeist.

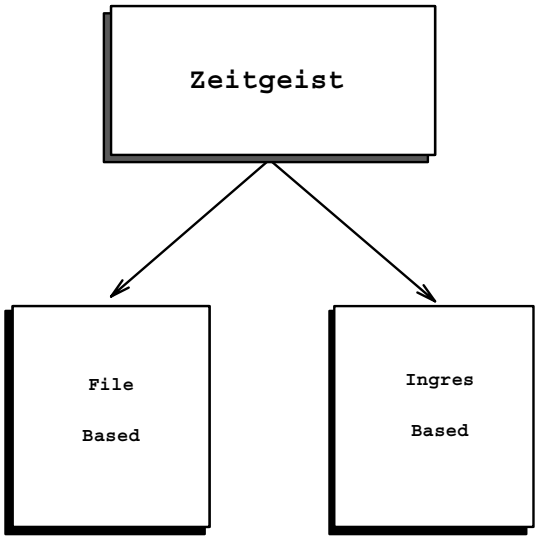

Figure 6.1. Storage Interface

6.2 Structure of Persistent Storage

The data stored in the persistent storage is in the form of three main relations in the underlying relational database. The first of these tables **Groups** contains the information for each storage group and the time when it was last modified. The storage group 0 is used to keep the time of the last access to the database. Validity checking for an existing storage group is also done with the data present in this relation. A storage group cannot be specified if it is not present in this table. When an object is designated for persistent in a storage group, the storage group number must exist in this table. If it does exist, then the valid object numbers for this particular storage group are also collected. The tuple for the storage group specifies the starting number for the valid object numbers.

The second relation **Value** is the main table where most of the information about the instance of an object is stored. A description of the attributes in this table is provided. Each object has a unique object identifier referred as oid. This in turn contains three sub-fields which actually make the oid unique. They are: storage group

number, object number and timestamp. Therefore, to uniquely identify an object we need to have all of these values. A default storage group number is provided if the user does not explicitly specify one. The object number specifies a unique number assigned to a particular object, and the timestamp is the system time. The next attribute *seq_value* is used to construct the whole encapsulated object. This comes into effect in case the information contained in the encapsulated object (translated) is too large for the underlying storage manager to handle in one instance. Then the information is broken down into suitable chunks or pieces to be stored conveniently. The *seq_value* specifies as to which part of the original data does this piece refer to. The *objsize_value* gives us the full size of the object information and not just the size of the chunk.

The attributes for the architecture oid and the type oid are made up of the same three sub-components as the object id and are currently not used in the implementation of Zeitgeist. Since initially there was some use envisaged for these, they were made as a part of the encapsulated object structure. Currently, NULL values are stored for them and passed around and there future use is highly unlikely.

The number of external references for each object are provided by the *extcnt_value* attribute. This gives the total number of objects referred to by this one. The actual data value for each object is finally stored in *bucket_value*. As explained earlier the size and piece number of this information is available and the whole object can be constructed.

The last of the relations, **Refto** is used to keep the external references of objects. The previous chapter lists the details. All those objects which are independently persistent, IPOs, are stored in this relation. These independent persistent objects can be accessed directly from Zeitgeist using the PTR call available. They can also be indirectly referred from the objects in their superclass. This table contains six

attributes. The first three form the oid for that object. *Seqno* gives the sequence number of the externally referenced object and *Xsgno* and *Xobno* give the storage group number and the object number for the object which is referencing these.

6.3 Using Embedded SQL

Embedded SQL was used to pass information from the C and C++ structures present in Zeitgeist to the underlying storage. Since this allows C variable names to be used directly to store and retrieve data via SQL, the object retrieved was build and passed to the Object Manager conveniently. Similarly, for storage of an encapsulated object the structure passed was directly mapped to SQL.

The object manipulation (and translation) takes place at the Object Manager level. Two independent buffers, for the Value and Refto relations, are maintained by the Object Manager. The maximum size of these buffers is currently 256 for Refto and 128 for the Value table. Whenever an object is designated as persistent, it is translated and put into this cache. This is done to minimize disk I/O. It would be highly uneconomical in terms of time to store each object on the disk as soon as it is made persistent. This disk I/O takes place only on two conditions. We keep track about the availability of memory on the cache. If the object are large enough and the buffer is full, all this data is written on the storage and the cache is flushed. The data is also written to the disk on an explicit *commit_transaction* call from the system or the user. In many cases, the transaction log space available on the underlying storage may not be large enough to store the whole cache. This can be due to many reasons. Even if the log space is initially vary large, operations by multiple users can limit the availability to Zeitgeist. In such cases, the Transaction Manager of the underlying storage aborts the transaction, resulting in the wrong message being conveyed to

Zeitgeist apart from losing data. This calls for an explicit *commit* to the underlying storage which suspends all operations and writes data to disk.

Recovery and concurrency control is also managed by the underlying storage. Zeitgeist by itself does not keep a log and facilitate recovery. All of the recovery is handled by Ingres. The recovery management changes with the storage mechanism. Concurrency is handled at two levels. A multi-level transaction model comes into picture here. One level of abstraction is the concurrency control managed by Zeitgeist, and at a lower level, Ingres manages concurrency in the three tables it maintains. The user, however is concerned only with the top-level, and sees concurrency only in the object-oriented paradigm, and not at the relational level. In the case of unexpected behavior by the system, calls to the recovery manager of Ingres are made. The SQL *command area (sqlca)* is extensively used for debugging and analyzing the internal details of the storage and disk I/O.

The database created in Ingres is also called Zeitgeist. This database is not available to the users directly. This is done to restrict access and direct operation on the translated data. Access to all potential users of the Zeitgeist front-end is provided by suitable grant permits to the Ingres database relations. All users do not have the need and hence the authority to reset or drop the tables and indexes present in the persistent storage. Resetting the tables flushes the whole Zeitgeist database and should be done with care. If the tables are dropped and recreated, grant permits to the users should be again provided. Currently, this is done by sql scripts and only one user, the DBA for Zeitgeist, is permitted such operations. When Zeitgeist is started up, the relational database is attached to the session. User authorization is checked at the time of attach to validate modifications to the database. Time entry in the Groups table is also validated to make sure that the storage was not accessed and modified externally. In case it was, the current time is taken from the system,

and the Groups table is updated accordingly. The cache where data for the Value and Refto tables is kept is also initialized at this time.

When a Zeitgeist transaction is committed, a symbolic name is allocated for objects. This allocation of a symbolic name is also done at Zeitgeist startup. A number of object numbers within a specified storage group are reserved. The call to allocate a symbolic name is made when an object is to be made persistent, and the Object Manager calls for a symbolic name to be associated in the persistent storage with this storage group. The number of storage group and the total number of objects desired in that storage group is passed. If the call was successful, then the first object number allocated is passed, otherwise 0 is returned. This information for continuity of object numbers is kept in the Groups table, which gives the starting number for the sequence of object numbers. Since the total count desired is also specified, the table is updated again for the next call to allocate symbolic names. This commit call also writes to the persistent storage and reinitializes the cache for further use. The sql COMMIT call is given to Ingres.

The *zgdb_putobject* procedure stores the data uniquely identifying a particular object into the cache. The object descriptor is passed. This object descriptor is a structure containing data to be stored in the value table. The encapsulated object can be completely built from this object descriptor. The legality of the object number in the storage group is again checked, before modifying the persistent storage. Since the translated data can be larger than what the persistent storage accepts for the *value* attribute, the object is partially broken into pieces for storage. If the size is less than that declared to the storage this splitting need not be done and the whole object can be inserted into the cache and later into the relation in one instance. Otherwise, suitable chunks, in sequence, are formed. All available external references for this object, also have to be inserted. For each of the external references a tuple is

inserted into the Refto relation. The timestamp, which is always zero, is not stored. The data, as yet, has not been stored into the persistent storage. It is still present in the Zeitgeist cache.

Another procedure of interest is the *zgdb_getobject*. This accesses the database and puts the data in the value buffer. To uniquely identify an object the oid is passed. The object is then constructed from this cache and the object descriptor is returned. This returned object has been built with the actual timestamp. Once the initial object description is available, more pieces of value are fetched. Now since we have the object, all the external references for this particular object are fetched. Once these references have been fetched into the cache, access to the persistent storage by other users can be permitted. Therefore an explicit commit call to Ingres is made, which releases the locks and all other pending stuff.

Initially, the *begin_transaction* call sets up the user for a commit. It flushes the buffers, both refto and value, and visualizes a successful completion. While initiating an *abort_transaction* call, an explicit ROLLBACK of work is made for Ingres. It aborts an in-progress commit, canceling all pending database writes and releasing all locks.

CHAPTER 7
CONCLUSION AND FUTURE WORK

Efficient and systematic evaluation of rules plays an integral role in the Active databases. Numerous database identifying themselves as *Active* have been proposed. However, most of these are passive in some sense of triggers being defined on events. Also, the restriction on the number of triggers limits the capabilities of these databases. The HiPAC design of various coupling modes and their execution is very promising. A suitable nested transaction model needed to be designed to accommodate the concept of different modes and the concurrent execution of rules.

In the thesis, a flexible and versatile model for the nested transactions has been presented. Changes in the prototype system, Zeitgeist have been proposed to incorporate the nested transaction model. Since the system was recently acquired, modifications also had to made in the Storage Manager to make it functional on our platform. A Storage Manager to support concurrent users was interfaced. This required rewriting the interface module to use Ingres as the Storage Manager, as opposed to using Oracle earlier.

With the system functional, the Transaction Manager was examined to incorporate the nested transaction model. A design has been presented, which uses the same functionality as the current TM with some modifications. Various implementation issues, such as data structure for the transactions and subtransactions, accessibility and data stored in the lock tables, searching costs have been extensively examined to make the actual implementation feasible. Two algorithms have been presented

to acquire and release locks in the model, taking into account the inheritance of information by the parent transaction from its inferiors and vice-versa.

Implementation of the proposed model and algorithms constitutes part of the on-going research activities in the Sentinel project. For more robust implementation, the DDL needs to be modified to support user defined subtransactions. Current design assumes more subtransactions to be spawned at the system level, at the time of rules being triggered. What if the user envisages concurrency in his application ? He should have the flexibility to spawn as many subtransactions as desired, without interfering with the system structure. Object state restoration in Zeitgeist non-existent. More research is warranted on the recovery methods to be used. Recovery is not supported by Zeitgeist independently, but is handled by the underlying Storage Manager. One of the main advantages of nested transactions is its finer recovery capability. This advantage needs to be fully exploited after the implementation is over.

APPENDIX

STRUCTURE OF RELATIONS USED IN INGRES

In this chapter the details of the relations used in Ingres are given. The embedded SQL declaration is presented and the C structure used to directly store and retrieve data from these relations is shown. With the description of each table more information is also presented. The various attributes and storage structure, type, pages etc. give an insight into the persistent store. Data from the relations is directly stored into the structures, as facilitated by the Ingres pre-processor for the C language. The file containing the host language (C) commands as well as the embedded SQL commands is pre-processed. The **.c** file generated is then compiled with the rest of Zeitgeist code linking the Ingres libraries as well.

```
/* Description of table groups from database Zeitgeist */

EXEC SQL DECLARE groups TABLE
        (sgno           integer not null,
        nextno          integer not null,
        loc             char(4),
        user_comment    char(50)) ;
struct groups_ {
        long            sgno ;
        long            nextno ;
        char            loc[5] ;
```

```
        char            user_comment[51] ;
        } groups ;


            /* Description of table refto from database Zeitgeist */


EXEC SQL DECLARE refto TABLE
        (sgno       integer not null,
        obno        integer not null,
        time        integer not null,
        seqno       integer not null,
        xsgno       integer not null,
        xobno       integer not null) ;
struct refto_ {
        long        sgno ;
        long        obno ;
        long        time ;
        long        seqno ;
        long        xsgno ;
        long        xobno ;
        } refto ;


            /* Description of table value from database Zeitgeist */

EXEC SQL DECLARE value TABLE
        (sgno       integer not null,
```

```
        obno        integer not null,
        time        integer not null,
        seqno       integer not null,
        objsize     integer,
        asgno       integer,
        aobno       integer,
        atime       integer,
        dsgno       integer,
        dobno       integer,
        dtime       integer,
        sacnt       integer,
        uacnt       integer,
        xcnt        integer,
        value       char(260)) ;
struct value_ {
        long        sgno ;
        long        obno ;
        long        time ;
        long        seqno ;
        long        objsize ;
        long        asgno ;
        long        aobno ;
        long        atime ;
        long        dsgno ;
        long        dobno ;
        long        dtime ;
```

```
long        sacnt ;
long        uacnt ;
long        xcnt ;
char        value[261] ;
} value ;
```

REFERENCES

[1] Philip A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys*, 13(3):185–221, July 1981.

[2] S. Chakravarthy and others. HiPAC: A Research Project in Active, Time-Constrained Database Management, Final Report. Technical Report XAIT-89-02, Xerox Advanced Information Technology, Cambridge, MA, Aug. 1989.

[3] U. S. Chakravarthy. Rule management and evaluation: An active DBMS perspective. *ACM Sigmod Record, special issue on rule processing in data bases*, 18(3):20–28, Sept. 1989.

[4] M. Darnovsky and J. Bowman. *TRANSACT-SQL User's Guide*. Document 3231-2.1, Sybase Inc., San Jose, CA, 1987.

[5] U. Dayal and others. The HiPAC project: Combining active databases and timing constraints. *SIGMOD Record, special issue of real time data base systems*, 17(1):51–70, March. 1988.

[6] K. P. Eswaran. Specifications, Implementations, and Interactions of a Trigger Subsystem in an Integrated data base system. IBM Research Report RJ1820, Almaden, Aug. 1976.

[7] K. P. Eswaran and D. D. Chamberlain. Functional specifications of a subsystem for data base integrity. *Proceedings 1st International Conference on Very Large Data Bases*, 110–124, Cannes, Sept. 1975.

[8] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in database systems. *Communications of the ACM 19*, 10(11), Nov. 1976.

[9] J. Grant. Normalization of relational database theories by adequate decompositions. Technical Report TR-1001, University of Maryland, College Park, Jan. 1981.

[10] J. N. Gray. Notes on Database Operating Systems: Lecture Notes in Computer Science 60, Springer-Verlag, New York, 1978.

[11] T. Haerder and A. Reuter. Principles of transaction oriented database recovery. *ACM Computing Surveys*, 15(4):287–318, Dec. 1983.

[12] T. Haerder and K. Rothermal. Concepts for transaction recovery in nested transactions. *Proceedings of SIGMOD*, 12(2), July 1987.

[13] InterBase Software Corporation, *InterBase DDL Reference Manual, InterBase Version 3.0*,Bedford, MA, 1990.

[14] V. Krishnamurthy, S. Y. W. Su and H. Lam. An Object-Oriented Semantic Association Model (OSAM*). *Theoretical Issues and Applications in Industrial Engineering and Manufacturing*, pages 242–251, 1989.

[15] Daniel P. Miranker. TREAT: A better match algorithm for AI production systems. *Proceedings of AAAI 87 Conference on Artificial Intelligence*, pages 42–47, Aug. 1987.

[16] J. Moss. Nested Transactions: An Approach to Reliable Distributed Computing. MIT Laboratory for Computer Science, MIT/LCS/TR-260, Cambridge, MA 1981.

[17] Edward Perez and Robert W. Peterson. Zeitgeist Persistent C++ User Manual. Information Technologies Laboratory Technical Report 90-07-02, 1991.

[18] L. Raschid and S. Y. W. Su. Transaction-oriented mechanism to control processing in a knowledge base management system. *Proceedings 2nd International Conference on Expert Database Systems*, pages 163–174, Portland, Oregon, 1988.

[19] T. Sellis, C. Lin, and L. Raschid. Data intensive production systems: The DIPS approach. *SIGMOD Record*, 18(3):42–50, Sept. 1989.

[20] T. Sellis and N. Roussopoulos. Deep compilation of large rule bases. *Proceedings 2nd International Conference on Expert Database Systems*, April. 1988.

[21] S. Y. W. Su. SAM*: A semantic association model for corporate and scientific-statistical databases. *Information Sciences*, 29:151–199, 1983.

[22] J. Widom, R. J. Cochrane, and B. G. Lindsay. Implemented set-oriented production rules as an extension of Starburst. *Proceedings 17th International Conference on Very Large Data Bases*, pages 275–286, Barcelona, Sept. 1991.

[23] J. Widom and S. Finkelstein. Set-oriented production rules in relational database systems. *Proc. of ACM-SIGMOD*, pages 259–270, May 1990.

BIOGRAPHICAL SKETCH

Asheesh Sharma was born in Nagpur, India. He received his undergraduate degree in electronics and communication engineering from Regional Engineering College, Bhopal, India, in May 1989. He will receive his Master of Science degree in computer and information sciences from the University of Florida Gainesville, in August 1992. His current research interests are distributed data bases, object-oriented data bases, and concurrency in transaction processing systems.