University of Florida

Computer and Information Science and Engineering

Realizing Transaction Models: An Extensible Approach using ECA Rules

E. Anwar
S. Chakravarthy
M. Viveros

EMAIL: sharma@cis.ufl.edu

WWW: http://www.cis.ufl.edu/~sharma

Tech. Report UF-CIS-TR-95-029
(Submitted for publication)

October 1995

(This work is partly supported by the Office of Naval Research and the Navy Command, Control and Ocean Surveillance Center RDT&E Division, and by the Rome Laboratory.)

Computer and Information Science and Engineering Department
E301 Computer Science and Engineering Building
University of Florida, PO Box 116120
Gainesville, Florida 32611-6120

Realizing Transaction Models: An Extensible Approach using ECA Rules*

E. Anwar† **S. Chakravarthy**

Database Systems Research and Development Center
Computer and Information Science and Engineering Department
University of Florida, Gainesville FL 32611
email: {emsa, sharma}@cis.ufl.edu

M. Viveros
IBM T. J. Watson Research Center
Hawthorne, New York 10532
email: viveros@watson.ibm.com

Abstract

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

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

1 Introduction

Conventional database management systems (DBMSs) guarantee atomicity, consistency, isolation and durability for each transaction (commonly referred to as the ACID properties) [GR93]. The emergence of non-traditional applications such as workflow management, cooperative tasks, and

*This work is partly supported by the Office of Naval Research and the Navy Command, Control and Ocean Surveillance Center RDT&E Division, and by the Rome Laboratory.

†Part of this work was performed during the author's internship at T. J. Watson Research Center, Hawthorne, NY.

computer integrated manufacturing (CIM), has made it apparent that the traditional transaction model is too restrictive for these applications. As an example, in a workflow application, some of the (sub)tasks that deal with invoices may have to satisfy the ACID properties (on a small portion of the database) whereas other tasks may work on their own copy of the data objects and only require synchronization.

The current solution for meeting the diverse requirements of these applications has been the proposal of advanced or extended transaction models such as nested transactions, Sagas, ConTract model, and Flex transactions [Mos81, GMS87, Reu89, ELLR90]. These transaction models relax the ACID properties in various ways to better model the parallelism, consistency, and serializability requirements of non-traditional applications. Despite their benefits, the proposed transaction models are *application-specific*. In other words, each model serves the requirements of a *particular class* of applications. This trend is likely to continue as it is improbable that a single transaction model can serve the requirements of all applications, as these requirements are diverse in nature. Consequently, since a DBMS typically supports *only one* transaction model, a DBMS can only serve the requirements of a particular class of applications. Therefore, it is critical that the solution to this problem aims at a framework which readily supports multiple transaction models, as well as support them on the same DBMS. Choice of a transaction model is usually based on application needs and is best made *at runtime*, not at database development/configuration time. This approach, if successful, will obviate the need for developing DBMSs suited for specific application classes. It is equally important to avoid hardwiring the semantics of all known transaction models, as this increases runtime checking as well as the footprint of the transaction manager.

Several frameworks including [Tra91, ASRS92, BP95, Moh94, BDG+94, GHKM94] have been proposed for supporting various transaction models. Our approach differs from current approaches in that we use the active database paradigm as a mechanism for supporting extended transaction models in a novel way. Our approach also models and enforces *auxiliary* semantics (other than those defining transaction semantics) useful for a number of applications within the same framework. For example, to reduce the possibility of rollbacks and unnecessary waits by transactions, it might be necessary to define semantics which specify thresholds on the number of long-lived transactions in the system.

Contributions

This paper proposes a uniform framework for realizing an extensible transaction management system. We introduce a different and novel use of the *active database paradigm* as a mechanism for specifying and supporting various transaction models in a DBMS. We use ECA rules defined and (efficiently) supported at the *system level* to customize the internal behavior of a database management system[1]. This is in contrast to the conventional approach of using rules at the *application level* to customize application behavior. We believe that various system functionality including fine tuning, transaction semantics, and index management, can be supported in a uniform manner using system level active capability. In this paper, we demonstrate the use of this approach only for realizing a number of transaction models. In order to accomplish this, it is important that the DBMS supports active capability in a way that allows the definition of ECA rules on system operations in an efficient manner. In this paper, we focus on the concurrency control aspects of

[1] There exists an extensive body of research on database extensibility; some of the proposed techniques can be applied to transaction management (e.g., [Tra91, Bat88]). However, most of the approaches have concentrated on its applicability to other database functionality such as access methods, optimization and data types. In contrast, transaction management extensibility has so far received little attention.

transaction management and the uniform framework for capturing both application and transaction model semantics. Furthermore, for simplicity, we assume that strict serializability is enforced across transaction models unless an explicit mechanism such as delegation is used to override it.

We show how the *semantics* of various transaction models, including the traditional transaction model, nested transactions, Sagas and Split transactions, can be enforced using a set of ECA rules. We also demonstrate the approach used to identify and develop these rules as well as show their reusability. The stored rule sets form a pool of rules where each set describes the semantics of a particular transaction model. We describe a framework which allows the association of *any* rule set (from the pool of rules) with a transaction. Thus, during the course of a transaction's execution, its semantics will be realized by the set of ECA rules associated with it. Moreover, rules can be activated and deactivated *dynamically* (by using subscribe/unsubscribe, activate/deactivate, and enable/disable as elaborated in a later section) thereby enabling transaction semantics to be modified dynamically. This is an important requirement for testing the applicability of different transaction models for the same application as well as dynamically adapting transaction semantics in response to changes in system load and throughput requirements. Furthermore, our ability to capture the traditional transaction model in terms of rules provides us with a DBMS which does not have a built-in transaction model. This, in turn, allows us to readily enforce strict serializability across applications that use different transaction model semantics within each application.

Most importantly, the applications are not restricted to the transaction semantics provided by the ECA rules stored in the active DBMS. Specifically, it is possible for the DBC to define additional ECA rules on the *underlying data structures* in the DBMS (such as the transaction table and object table). This provides a powerful facility since the DBC can *build* or equivalently *construct* arbitrary transaction semantics rather than being restricted to those transaction models provided by the system. Furthermore, the framework provided in this paper separates the definition of rules (which define transaction semantics) from the application code itself. This permits applications and transaction semantics to be modified independently of each other as well as use of existing applications without major modifications. For concreteness, we show how various transaction models can be translated into a set of ECA rules in the context of Sentinel [AMC93, CKAK94, CKTB95], an active OODBMS developed at UF. However, our framework and approach [CA95] is general and can be applied to any DBMS supporting active capability at the system level (relational or otherwise).

The remainder of this paper is structured as follows. In section 2 we present our general approach for supporting transaction models using ECA rules. We also give a brief overview of Sentinel and show how its features are used to realize an extensible transaction management system. Section 3 follows with our design and implementation details. Section 4 discusses the extensibility of our approach while a brief overview of the state-of-the-art is presented in section 5. Our conclusions and future directions for research are included in section 6.

2 Our Approach

A transaction performs a number of operations during the course of its execution – some specified by the user and some performed by the system to guarantee certain properties. The *semantics* of the operations performed by the system differ from one transaction model to the other. For instance, the semantics of the *commit* operation in the traditional transaction model entails updating the log, making all updates permanent in the database, and releasing all locks held. This is in contrast

to the commit of a subtransaction (in the nested transaction model) where all locks are inherited by the parent and the updates *not* made permanent until all superior transactions commit. As another example, a transaction in the traditional transaction model can acquire an *exclusive-lock* on an object if no other transaction holds *any* lock on that object. This is different from the nested transaction model where a subtransaction may acquire an *exclusive-lock* on an object even if one of its ancestor transactions holds a lock on that object. Moreover, some transactions perform operations which are very specific to that transaction model (and not shared by other transaction models). As an example, in the Split transaction model, a transaction may perform the operation *split* which causes the instantiation of a new top-level transaction and the delegation of some uncommitted operations to it.

It is apparent that in order to support different transaction models in the *same* DBMS, one should not hardwire the semantics of operations such as commit, abort, read and write[2]. Instead, a flexible mechanism is needed for associating computations (ECA rules, in our case) with system operations, as well as with some operations performed by the system on behalf of users. Furthermore, for this mechanism to be effective and extensible, it should be independent of the programming model and the environment. And this is precisely what active capability *supported at the system level* offers. Moreover, the utility of active capability for supporting application specific behavior has been well established, as can be observed by the presence of this capability in almost all commercial models, its introduction into SQL3, and the number of research prototypes being developed. The availability of expressive event specification languages (e.g., Snoop, Samos, Ode, Reach) that allow sequence, conjunction and time related events can be beneficial for modeling some of the synchronization aspects of workflow and other transaction models. However, the presence of active capability at the application level does not guarantee that it can be used at the system level as well. To the best of our knowledge, this is true for most of the commercial and research prototypes of active database systems. Sentinel differs in this regard and supports both application level and system level active capability in a uniform manner. Sentinel is described in the following section.

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

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

2. The second step involves identifying the condition which needs to be evaluated when an event occurs (e.g., checking for conflicts at lock request time) and the action to be performed if the condition evaluates to true (e.g., granting the lock to the requesting transaction). The events, conditions and actions yield pools of events, conditions, and actions, respectively, which are stored in the DBMS. These pools, depicted in Figure 1, form the building blocks from which rules are constructed.

[2]Although support for different transaction models, to some extent, can be accomplished in an object-oriented environment by creating a transaction hierarchy and overloading the operations or methods, this approach is specific to the model used rather than the system.

3. The final step involves combining an event, a condition and an action to compose an ECA rule. Each ECA rule defines the semantics of a smaller unit of the transaction model under consideration. For instance, an ECA rule may define the semantics of the *acquire lock* operation. This process is repeated until a *rule set* defining the entire semantics of a transaction model, is built. We allow for the cascading of rule execution. This occurs when the *action* component of a rule raises event(s) which may trigger other rule(s). Cascading of rules is utilized for implementing nested transactions and is shown in section 3.1.

This approach allows sharing of the building blocks in several ways. Events, conditions, and actions are shared across rules sets composed for different transaction models. In addition, intermediate rules can also be shared by other rules. Although Figure 1 shows a single level for clarity, a hierarchy of rules is constructed from the building blocks. The overlap of events, conditions and actions for different rule sets clearly indicates the modularity and reusability aspect of our approach. This is further substantiated in the section on implementation details.

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

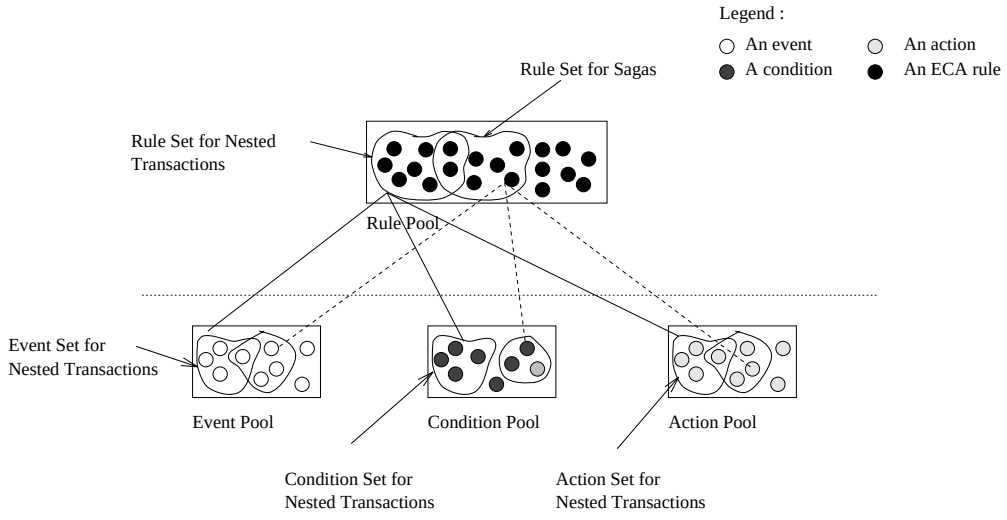

Figure 1: Rule Composition

The mechanism described above can also be applied for customizing auxiliary behavior. By trapping the operations that are executed by applications, it is possible to perform *auxiliary* actions as required by the user/system designer (i.e., other than those defining the transaction semantics). A good example of this is in systems where optimal performance is achieved when the number of transactions in the system does not exceed a particular threshold (e.g., load balancing and buffer sizes). Therefore, it is necessary to check the number of transactions and not allow the threshold to

be exceeded. This can be accomplished by trapping the operation *begin transaction* and checking the number of active transactions at that point. If the number is found to be less than the threshold, then allow the transaction to continue execution, otherwise either abort the transaction or make it wait. Similarly, in banking applications there may be a limit on the number or amount of withdrawals in a day. By defining a rule which is triggered upon detection of the *begin* operation, it is possible to check the number or amount of withdrawals appropriately and either continue or abort the transaction. To summarize, not only does the active database paradigm allow for the specification of transaction semantics but arbitrary semantics as well in an extensible manner.

2.1 Overview of Sentinel

Sentinel [CM94, AMC93, CKAK94, CKTB95] is an active object-oriented DBMS that seamlessly integrates ECA rules into the object-oriented paradigm. The Sentinel architecture is an extension of the *passive* Open OODB system architecture [OOD93]. The behavior of objects has been extended to facilitate support of both *system level* and *application level* active capability in a uniform manner. Specifically, objects, in Sentinel, are classified into: passive, reactive and notifiable objects. *Passive objects* are conventional objects which receive messages, perform some operations and then return results. *Reactive objects*, on the other hand, are objects that need to be monitored (i.e., on which rules will be defined). A reactive object can declare a set of *events* and when these events occur, notifiable objects associated with these events are informed. Lastly, *Notifiable objects* are objects that can be informed of the events produced by reactive objects. Therefore, notifiable objects become aware of a reactive object's state changes and can take appropriate measures (by evaluating conditions and executing actions).

Passive objects do not generate events. An object that needs to be monitored (by informing other objects of its state changes) cannot be passive. Reactive objects generate primitive events. These events can be combined to form complex events by using event operators (e.g., sequence, conjunction, aperiodic). Notifiable objects *subscribe* to the events dynamically. After subscription, the reactive objects propagate their generated primitive events to the notifiable objects. Events and rules are examples of notifiable objects. The event detector receives events from reactive objects, composes them, and rules are executed as specified.

Object classes for making Open OODB active are shown in Figure 2. The key feature of Sentinel is its ability to make the entire system active by making the oodb class (root class of the Open OODB system) reactive. In essence, active capability can be imparted to any class (either application defined or system defined) by the same mechanism. Furthermore, making a class both reactive and notifiable allows one to write meta-level rules, thereby changing the behavior of the rule system. This feature, for example, can be used to order rules that are associated with the same event.

The *subscription* mechanism of Sentinel allows the binding of condition-action pairs to events at runtime. It is possible to associate and disassociate as well as enable and disable rules (or rule sets) in Sentinel at *runtime*. This feature is critical without which it will not be possible to change execution semantics at runtime. This ability for late binding also reduces the amount of rules managed by the system at a given point in time and contributes to the efficiency of the system. The separation of objects into passive and reactive allows one to selectively determine object types. For example, instead of making the oodb class active, only the *Transaction_mgr* and *Lock_mgr* classes can be made active to improve performance. Classes are made active by deriving them from the *Reactive* class depicted in Figure 2.

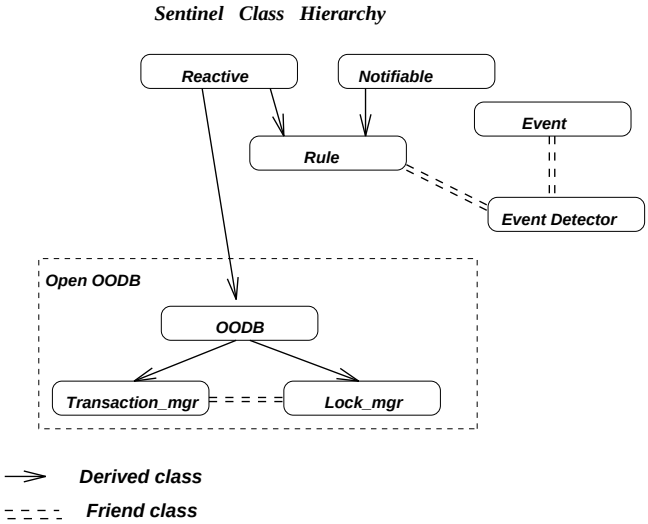

Figure 2: Sentinel class hierarchy

Currently, there are two versions of Sentinel. One version uses Exodus as the storage manager and has a client/server architecture. The other version, Zeitgeist, is an earlier version of Open OODB and uses Oracle or Ingres as a pure storage manager. In the client/server architecture, Exodus acts both as a storage manager and as a transaction manager providing both concurrency control and recovery for top level transactions [3]. However, in Zeitgeist, Ingres or Oracle act only as a storage manager and the transaction manager (without recovery) is divorced from the storage manager. As our current focus is in demonstrating the extensible support for transaction models using system level active capability, we use Zeitgeist as the implementation platform. Otherwise, Exodus has to be made active prior to incorporating our rules shown in the rest of the paper. Our current implementation uses the data structures supported in Zeitgeist, as that fits well with the scope of current work.

2.2 Benefits of Our Approach

Our approach – Sentinel functionality and the rule composition process – provides the following benefits:

- Applications can avail the semantics of a particular transaction model T_x by *enabling* the rule set defining the semantics of T_x. Disabling a rule set will eliminate runtime overhead associated with the firing of rules in that rule set.
- Using the proposed approach, different applications can realize desired transaction semantics by *choosing* the appropriate rule set. It is also possible to make individual rules available to applications by the DBC in an appropriate manner. This usage requires a good understanding of the internals and hence needs to be used cautiously.
- More importantly, realizing the semantics of a particular transaction model entails defining a rule set to be used by transactions adhering to this model. As *reusability* of rule sets among

[3] We also have a 2 level implementation of the nested transaction model in the client/server architecture[Bad93].

different transaction models is a key aspect of this approach, it may be possible to define a new rule set using existing rules.

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

- It is possible to *configure* a DBMS with one or more rule sets and optimize the rule sets for efficiency. In other words, efficiency need not necessarily be sacrificed if only one or a small set of transaction models are desired. This can be achieved by using the subscribe/unsubscribe functionality, or by compiling desired rules at configuration time [LS95]. Subscribe and unsubscribe allows us to decouple rules (condition-action pairs) from events, thereby reducing runtime overhead.

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

3 Implementation Details

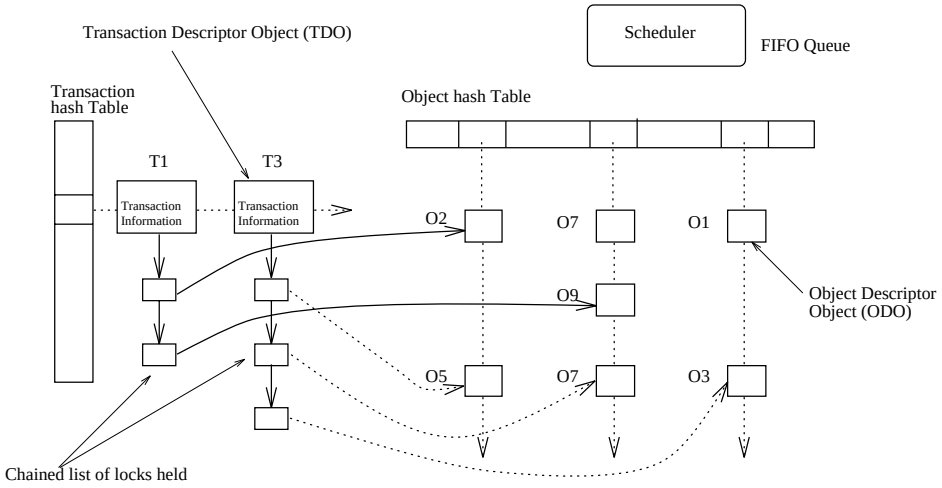

Figure 3: Data structures pertinent to Transaction Processing.

This section presents detailed design and implementation aspects of ECA rules for supporting various transaction models. Our implementation assumes the ECA rule functionality provided by Sentinel [AMC93, CKAK94, CKTB95], an active OODBMS developed at UF. Although the details are presented in the context of an OODBMS, the approach is independent of the database model as explained earlier.

Realizing various transaction models using our approach entails *trapping* the operations performed by transactions (both user visible and internal operations) as well as associating one or more rules with each operation. As the execution of a transaction invokes operations on the underlying data structures, it is necessary to trap some of these operations. The rules presented in this section

```
class Transaction_Descriptor  : Reactive              // By deriving this class from  the Reactive class it is  possible
{                                                     // to trap the invocation  of the methods in this class
    public :
        long   tid;     // transaction identifier
        PFVOID transaction_body;     // pointer to function containing transaction body
        TYPE transaction_type;       // this denotes the type of the transaction, e.g., TOPLEVEL, CHILD etc.
        STATUS transaction_status;     // the status of the transaction, e.g., ACTIVE, COMMITTED, etc.
        LIST_OF_OBJ_DESC   *locks;    // pointer to a chained list of object descriptor objects, i.e., lock informationn
        Transaction *next;      // pointer to next transaction object in bucket

        Transaction_Descriptor();   // class constructor
        ~Transaction_Descriptor();   // class desctructor
        long Initialize_Descriptor(PFVOID tx_body, TYPE tx_type);    // initializes object and returns transaction identifier
        begin();      // semantics of this method depends on enabled rules
        commit();       // semantics of this method depends on enabled rules
        abort();      // semantics of this method depends on enabled rules
        delegate(SET_OF_OBJS objs, long Tx_id);     // delegate locks and operations on objs to transaction Tx_id
        acquire_lock(long oid, MODE lock_mode);   // semantics of granting lock depends on enabled rules
        release_locks();    // semantics of this method depends on enabled rules
        upgrade_lock(long oid, MODE new_mode);    // upgrades lock held
        read_only(long oid);     // this fetches the object from disk in read only mode, i.e., RO-mode
        read(long oid);      // this fetches the object from disk in either shared or exclusive mode
        add_lock(long oid, MODE mode);    // appends this object descriptor object to end of lock list held by transaction
};
```

Figure 4: The Transaction_Descriptor Class.

focus on concurrency control and functionality issues. Performance and recovery issues are being addressed separately and are beyond the scope of this paper. We have chosen commonly used data structures [GR93] to keep the description simple and easy to understand. The data structures, on which operations need to be trapped, are depicted in Figure 3 and are the *transaction table, the object table* and *scheduler*. Below, we briefly describe the structure of each one and later provide the actual classes which implement them.

Transaction Table: This is a hash table that maintains information about transactions submitted to the DBMS. Each hash table bucket points to a list of transaction descriptor objects (TDO). Each TDO contains the transaction's tid, a pointer to a function representing the transaction body, the type of the transaction (e.g., top-level, child, sagas), the transaction status, a chained list of the locks held, and a pointer to the next transaction in the bucket. Maintaining a chained list of locks held facilitates the fast of release of locks at commit time.

Object Hash Table: This data structure is also a hash table that maintains lock information on objects currently used by transactions. Each bucket points to a list of object-descriptor objects whose oid's have hashed to the bucket. Each object-descriptor object (ODO) contains the oid of the object, the lock mode it was granted in, a counter indicating the number of transaction object's holding the lock on the object (i.e., the number of transaction objects pointing to this ODO), and a pointer to the next ODO in the bucket.

Scheduler: This data structure is basically a FIFO queue of transaction bodies to be executed.

The transaction hash table and object hash table are used together by the DBMS as depicted in Figure 3. Each transaction maintains a chained list where each item in the list points to an

```
class Object _Descriptor  : Reactive                  // By deriving this class from the Reactive class it is possible
{                                                     // to trap the invocation of the methods in this class
    public :

        long  oid;    //object identifier
        MODE  lock_mode;    // lock mode held on object , e.g., READONLY, SHARED, EXCLUSIVE, RETAINED, etc.
        int counter;     // number of transactions holding a lock on this object
        Object_Descriptor *next_in_bucket;  // pointer to next object descriptor which hashes to this bucket


        Object_Descriptor(long oid, MODE mode);      // constructor which creates object descriptor object
        ~Object_Descriptor();             // class destructor
        decrement_counter;                // decrements the counter attribute by 1
};
```

Figure 5: The Object_Descriptor Class.

ODO in the object hash table. This chained list represents the objects held by a transaction. For example, transaction T1's chained list is the solid line illustrated in Figure 3. Here, transaction T1 holds locks on objects O2 and O9. Similarly, the locks held by transaction T3 are represented by the dashed lines and show that T3 holds locks on objects O5, O7 and O3. It is important to note that more than one transaction can point to the same object in the object hash table. This arises when the object is held either in read-only or shared mode by several transactions. Each ODO maintains a counter denoting the number of TDO's pointing to it[4].

It is important to note that an object may be replicated within a bucket. For example, there are two ODO's for object 07 in the second bucket depicted in Figure 3. This situation arises in the nested transaction model, specifically when a child acquires a lock on an object already held by one of its ancestor transactions. Replicating the ODO in this situation is necessary since there is only one place holder in the ODO denoting the lock mode and a child may hold the lock in a different mode than that held by its ancestor transaction. The classes implementing TDO's and ODO's are shown in Figures 4 and 5, respectively. The methods of these classes represent some of the *significant* events which need to be trapped to realize various transaction semantics. It should be emphasized that the DBC can modify these class interfaces to support different or new transaction models as they emerge.

In the following sections we briefly describe the semantics of nested transactions, Sagas and Split transactions and show the ECA rule sets necessary for modeling each. High-level specifications of ECA rules are given in order to enhance readability. The reader is referred [CA95] to for *the high-level support* provided to users for enabling rule sets in application programs. For brevity, we use RO-mode, S-mode and X-mode to denote locks in read-only, shared, and exclusive modes, respectively.

[4]It is possible to replicate an ODO in the hash table when multiple transactions hold it in either read-only or shared mode. In the interest of efficient main memory utilization (hash tables reside in main memory), we did not opt for this.

3.1 Modeling Nested Transactions

In the nested transaction model [Mos81], a transaction may contain any number of subtransactions, and each subtransaction, in turn, may contain any number of subtransactions. Hence, the entire transaction forms a hierarchy of transactions the root of which is called the *root* or *top-level* transaction. Transactions having subtransactions are called *parents*, and their subtransactions are their *children*. The transactions on the path from a transaction to the root of the transaction tree are called the *superiors* of the transaction. The nested transaction model allows several types of concurrency: sibling concurrency, parent/child concurrency, and the most general case – complete concurrency. We focus on sibling concurrency as it is the most widely used nested transaction model.

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

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

Rule: Tx_initiate

On `T1->Transaction_Descriptor::begin()` *// detecting invocation of begin method*
Condition `True` *// no condition checking necessary*
Action `sched->Scheduler::Insert(T1->tid)` *// Place transaction on scheduler queue*

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

Rule: Tx_Release_All_Locks

On `T1->Transaction_Descriptor::release_locks()` *// detecting release_locks method*
Condition `True` *// no condition checking necessary*
Action *// start releasing all locks held*

```
trav = T1->locks;       // point to head of lock list held by T1
while(trav != NULL) {
   get_exclusive_sem(); // get exclusive semaphore to access shared data
    trav->ODO->counter--; // decrement no. of transactions pointing to ODO
    temp = trav->ODO;     // make a temporary variable point to ODO
    trav = trav->next_obj_desc; // traverse to next lock held by transaction
    if(temp->counter == 0) // check if no transactions pointing to ODO
```

```
        free(temp);       // release memory used by ODO

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

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

Rule: Tx_commit_TopLevel

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

Rule: Tx_commit_Child

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

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

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

Rule: Tx_abort_TopLevel

```
On T1->Transaction_Descriptor::abort()      // detecting invocation of method abort
Condition T1->transaction_type == TOPLEVEL      // T1 is a top-level transaction
Action
      Flush buffers      // discard all changes made to objects
      Raise release_locks event      //this triggers rule Tx_Release_All_Locks
```

Rule: Tx_abort_Child

```
On T1->Transaction_Descriptor::abort()      // detecting invocation of method abort
Condition T1->transaction_type == CHILD      // T1 is a child transaction
Action
      Flush buffers      // discard all changes made to objects
      Raise release_locks event      //this triggers rule Tx_Release_All_Locks
```

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

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

Rule: Tx_grant_lock

```
On T1->Transaction_Descriptor::acquire_lock(oid,mode) // detecting add_lock method
Condition True      // no condition checking necessary
Action
// Create a new ODO, insert it in object hash table, & add it to list of locks held by T1
      Object_Descriptor* ODO(oid,mode);      // create new ODO
      i = hash(oid);      // find bucket to insert new ODO
      get_exclusive_sem();      // get an exclusive semaphore to modify object hash table
      object_table[i].insert(ODO);      // insert ODO in bucket
      Make T1 point to new ODO in bucket      // insert ODO at end of lock list held by T1
```

```
release_exclusive_sem();      // release exclusive semaphore
```

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

Rule: Tx_acquire_exclusive_lock_TopLevel

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

Condition

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

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

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

        return(1);
```

Action

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

Rule **Tx_acquire_lock_Child** defines the semantics of lock acquisition in all modes for nested transactions. A transaction is allowed to proceed if it already holds the lock in the requested mode

or the transaction holding the lock in conflicting mode is an ancestor transaction. Otherwise the subtransaction is blocked on a semaphore until it can acquire the lock.

Rule: Tx_acquire_lock_Child

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

Condition *// check transaction type & locking rules*

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

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

Action

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

4 Extensibility

This section demonstrates the extensibility of our approach. Specifically, we show how *existing* rules can be reused to express the semantics of other transaction models. Split transactions and Sagas are used as a basis for illustrating this.

4.1 Split Transactions

Split transactions [PKH88], were proposed mainly for supporting open-ended applications. In this transaction model, a transaction can execute the operation *split-transaction* which basically creates a *new* top-level transaction. The original transaction and the new transaction are serialized as if they are two independent transactions. However, when the original transaction executes the operation *split-transaction*, it can delegate responsibility of uncommitted operations on a *specific* subset of objects to the newly created transaction. After the split occurs, the two transactions continue execution and commit or abort independently. Similarly, a transaction can also execute

the operation *join-transaction* which essentially combines two active serializable transactions into one transaction. The main advantage of split transactions is relaxing isolation which is achieved when either the original or new transaction commits and releases its results.

One approach for supporting split transactions is to write ECA rules expressing their semantics using the three step process described in section 2. That is, identify the events, write new sets of conditions/actions, and combine them into rules. Although this approach yields a correct solution, it does not exploit reusability of rules among transaction models. A more beneficial approach is to examine currently defined ECA rules (i.e., events, conditions and actions defined for supporting various transaction models) and determine their reuse (either entire rules or components thereof) for expressing the new transaction model at hand. This allows one to understand the similarities as well as differences among transaction models, expedites the definition of the semantics of a transaction model (as rules may no longer need to be written from scratch), reduces the number of rules in the system, and most importantly provides extensibility.

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

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

Rule: Tx_split

On `T1->Transaction_Descriptor::split(obj_set, tx_body)` *// detecting invocation of split method*

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

Action

```
Transaction_Descriptor *New_Tx;      // create new transaction descriptor object
tid = New_Tx->Create_Descriptor(tx_body, TOPLEVEL); // initialize & get tid of
                                                    new toplevel transaction
∀ o_i such that o_i ∈ obj_set
    delegate(o_i, tid);      // delegate all locks & uncommitted
                             operations on o_i to tid
sched->Scheduler::Insert(T1->tid)   // Place new transaction on scheduler queue
```

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

Rule: Tx_join

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

Condition `Tx->transaction_status == ACTIVE` *// transaction to join with is active*
Action

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

4.2 Sagas

Sagas [GMS87] is a transaction model introduced to more adequately serve the requirements of long-lived transactions. A Saga consists of a set of independent component transactions $T_1, T_2, ..., T_n$ where each component transaction T_i (except transaction T_n) has an associated compensating transaction CT_i. Compensating transactions *semantically undo* the effects of their respective component transaction. The component transactions $T_1, T_2, ..., T_n$ execute serially in a predefined order and may interleave arbitrarily with the component transactions of other sagas. If a component transaction aborts, then the entire Saga aborts by executing the compensating transactions in reverse order to the order of the commitment of the component transactions. Here, we do not show the ECA rules which define the semantics of this transaction, but rather discuss how to modify the ECA rules defining nested transactions to achieve the semantics of this model.

Component and compensating transactions are top-level transactions whose semantics for *begin*, *commit*, *abort* and *acquire_lock* are very similar to those of top-level transactions belonging to the nested transaction model. To elaborate, the semantics of *commit* in a top-level component transaction is to make all updates permanent to the database and release all locks held. This is precisely the semantics of commit by a top-level transaction in the nested transaction model. In addition, the commit of a component transaction should also begin executing the next component transaction in the series. Therefore, the action part of rule **Tx_commit_TopLevel** should be modified to include starting the execution of the next component in the saga series. Similarly, the abort of a component transaction performs the same operations as the abort of a top-level transaction in the nested transaction model. In addition, it also starts execution of the appropriate compensating transaction in order to start the rollback process. Thus the action part of rule **Tx_abort_TopLevel**, given in section 3.1, should be modified to reflect this difference.

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

5 Related Work

Several efforts including [BDG+94, GHKM94, BP95] have addressed the problem of supporting extended transaction models. ASSET [BDG+94] provides a procedural approach for realizing transac-

tion semantics in applications. Basically, a set of transaction primitives (e.g., *begin, commit, wait*) are provided along with their semantics. The semantics of different transaction models are created by synthesizing high-level constructs in terms of these primitives by invoking them at appropriate points in the transaction body. A sophisticated used could also use these primitives directly. The data structures and the algorithms used to sketch the implementation of these primitives were described in a modified version of the EOS storage manager. DOM [GHKM94] views extended transactions as complex transactions which consist of a set of constituent transactions and a set of dependencies between them. Dependencies are classified into state and correctness dependencies. DOM supports extended transactions by accepting dependencies and enforcing them. Dependencies are expressed using the provided transaction specification language. State dependencies are implemented by translating their specifications into ECA rules while correctness dependencies are implemented using traditional scheduler technology. [BP95] adopts a layered approach for implementing extended transaction models. Specifically, add-on modules, referred to as *transaction adapters*, are *built* on top of existing transaction processing (TP) monitor components, aiming at extending the component's functionality for supporting extended transactions. Each transaction adapter maintains the necessary data structures and operations on them for supporting extended transactions. Implementation of the transaction adapters is outlined using Encina [Tra91] as the underlying TP monitor.

Our approach is different from ASSET in that our approach is based on system level ECA rules. Unlike the ASSET approach, we realize the semantics of individual units work in a transaction model and build ECA rules. Our approach provides benefits both for the designers of the DBMS and the users of the system. Designers of the system can configure systems for a choice set of transaction models. Run time overhead is eliminated by disabling and enabling rule sets at run time. Furthermore, users of the system are no longer constrained to those transaction models which can be defined using the hardwired primitives; rules can be chosen/defined/enabled/disabled by the DBC at runtime to achieve a wide range of transaction models. In contrast to our detailed framework of rule usage, DOM has concentrated on the specification of dependencies and adequacy of the specification mechanism. ECA rule usage is discussed mostly at the conceptual level.

6 Conclusions

In this paper, we have taken an extensible approach to support extended transaction models. We have demonstrated a novel application of system level ECA rules and concomitant functionality required to support various transaction models. We have analyzed several extended transaction models to derive detailed ECA rules (using low level data structures) necessary for modeling nested transactions (with sibling concurrency), Split transactions and Sagas. We have shown that by not hardwiring the semantics of operations such as commit, abort and acquire-lock, and detecting events (primitive and abstract) at runtime, it is possible to realize different transaction models. Our approach is extensible and can relatively easily support current transaction models as well as newer transaction models as they become available (by reusing existing rules as much as possible). The DBC can add or modify the class interface of the underlying data structures and define additional rules on its operations. In order to demonstrate the versatility of our approach, we have used data structures (on which rules were defined) that are similar to those found in most commercial transaction processing monitors. In particular, our approach assumes no specific underlying architecture or database model and can be applied to any active DBMS. A complete implementation is currently underway using Sentinel.

In this paper, we focused on addressing the concurrency control and functionality issues related to supporting various transaction models. We are currently investigating other related issues, primarily recovery, performance, and optimization of system level ECA rules.

References

[AMC93] E. Anwar, L. Maugis, and S. Chakravarthy. A New Perspective on Rule Support for Object-Oriented Databases. In *Proceedings, International Conference on Management of Data*, pages 99–108, Washington, D.C., May 1993.

[ASRS92] P. Attie, M. Singh, M. Rusinkiewicz, and A. Sheth. Specifying and enforcing intertask dependencies. Technical Report MCC Report: Carnot-245-92, Microelectronica and Computer Technology Corporation, November 1992.

[Bad93] R. Badani. Nested Transactions for Concurrent Execution of Rules: Design and Implementation. Master's thesis, Database Systems R&D Center, CIS Department, University of Florida, Gainesville, FL 32611, October 1993.

[Bat88] D. Batory. GENESIS: An Extensible Database Management System. *IEEE Transactions on Software Eng*, Nov. 1988.

[BDG+94] A. Biliris, S. Dar, N. Gehani, H.V.Jagadish, and K. Ramamritham. ASSET: A System for Supporting Extended Transactions. In *Proceedings, International Conference on Management of Data*, May 1994.

[BP95] Roger Barga and Calton Pu. A practical and modular method to implement extended transaction models. In *Proceedings, International Conference on Very Large Data Bases*, pages 206–217, Zurich, Switzerland, 1995.

[CA95] S. Chakravarthy and E. Anwar. Exploiting active database paradigm for supporting flexible transaction models. Technical Report UF-CIS TR-95-026, University of Florida, E470-CSE, Gainesville, FL 32611, June 1995.

[CKAK94] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim. Composite Events for Active Databases: Semantics, Contexts, and Detection. In *Proceedings, International Conference on Very Large Data Bases*, pages 606–617, August 1994.

[CKTB95] S. Chakravarthy, V. Krishnaprasad, Z. Tamizuddin, and R. Badani. ECA Rule Integration into an OODBMS: Architecture and Implementation. In *Proceedings, International Conference on Data Engineering*, Feb. 1995.

[CM94] S. Chakravarthy and D. Mishra. Snoop: An Expressive Event Specification Language for Active Databases. *Data and Knowledge Engineering*, 14(10):1–26, October 1994.

[ELLR90] A. Elmagarmid, Y. Leu, W. Litwin, and M. Rusinkiewicz. A multidatabase transaction model for Interbase. In *Proceedings of International Conference of Very Large Data Bases*, August 1990.

[GHKM94] D. Georgakopoulos, M. Hornick, P. Krychniak, and F. Manola. Specification and management of extended transactions in a programmable transaction environment. In *Proceedings IEEE Conference on Data Engineering*, February 1994.

[GMS87] H. Garcia-Molina and K. Salem. Sagas. In *Proceedings of the Conference on Database Systems in Office, Technique and Science*, pages 249–259, May 1987.

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

[LS95] F. Llirbat and E. Simon. Optimizing active database transactions: A new perspective. In *proc. of the 1st Int'l Workshop on Active and Real-Time Database Systems*, Skovde, Sweden, June 1995.

[Moh94] C. Mohan. Tutorial: A Survey and Critique of Advanced Transaction Models. In *Proceedings, International Conference on Management of Data*, page 521, Minneapolis, Minnesota, May 1994.

[Mos81] J. E. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing.* PhD thesis, Department of Electrical Engineering and Computer Science, MIT, 1981.

[OOD93] OODB. Open OODB Toolkit, Release 0.2 (Alpha) Document. Texas Instruments, Dallas, September 1993.

[PKH88] C. Pu, G. Kaiser, and N. Hutchinson. Split-transactions for open-ended activities. In *Proceedings, International Conference on Very Large Data Bases*, 1988.

[Reu89] A. Reuter. Contract: A means for extending control beyond transaction boundaries. In *Proceedings of the 2nd International Workshop on High Performance Transaction Systems*, September 1989.

[Tra91] Transarc. Encina product review. Technical report, Transarc Corp., 1991.

A Additional Rules for Nested Transactions

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

Rule: Tx_acquire_shared_lock_TopLevel

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

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

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

```
            Block transaction on semaphore;      // object is held in X-mode
            break;      // break out of loop & acquire lock once transaction is unblocked
         }

         return(1)
```

Action

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

Rule: Tx_acquire_readonly_lock_TopLevel

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

Condition

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

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

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

         return(1);
```

Action

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