Exploiting Active Database Paradigm For Supporting Flexible Transaction Model

S. Chakravarthy
E. Anwar

EMAIL: sharma@cis.ufl.edu

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

Tech. Report UF-CIS-TR-95-026

April 1995

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

Contents

Exploiting Active Database Paradigm For Supporting Flexible Transaction Models [*]

(Full Paper)

S. Chakravarthy **E. Anwar**

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

April 1995

Abstract

The emergence of non-traditional applications such as CAD/CIM, workflow management, and cooperating tasks reveal the limitations of the traditional transaction model. Specifically, the traditional transaction model (satisfying the ACID properties) is too restrictive - in terms of parallelism, consistency and serializability - for supporting the transaction requirements of these applications. The current solution to this problem has been the proposal of extended transaction models (e.g., long-lived transactions, nested transactions, Sagas and the DOM transaction model), which relax the ACID properties in various ways, to support these applications. However, this solution is inadequate primarily for two reasons: i) an increase in the number of new applications having varying transaction requirements leads to a proliferation of transaction models which are required for their support, and ii) this proliferation of transaction models will increase the difficulty of integrating the various models in a uniform manner into a DBMS.

In this paper, we make a case for using the active database paradigm for supporting *system functionality* (as opposed to the current use of enhancing the *user/application functionality*). We argue that this approach results in an extensible system in which it is relatively easy to add or modify the behavior of transaction models supported by the system. Furthermore, we demonstrate the use of ECA rules on system events in order to support newer transaction models. Finally, we show how our approach can be implemented in a DBMS that supports active capability. We use Sentinel - an object-oriented, active database system developed at UF – for illustrating the implementation choices.

1 Introduction

Traditionally, database management systems (DBMSs) guarantee atomicity, consistency, isolation and durability (commonly referred to as the ACID properties) [Gra81, HR83, OV91] for each

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

transaction. This transaction model, however, does not meet the requirements of a number of non-traditional applications where there may be a need for relaxing some of the ACID properties, and furthermore support a selective subset of the ACID properties. 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 require only synchronization. As another example, in design environments, compensating actions (or even partial rollbacks) may be more appropriate when a long running design activity reaches an undesirable point than aborting or rolling back a transaction to its starting point.

To overcome the limitations of the conventional transactions, a number of advanced or extended transaction models, such as nested transactions, Sagas, ConTract model, Flex transaction model, DOM transaction model etc. have been proposed in the literature [Mos81, GMS87, Reu89, ELLR90, BOH$^+$]. Some of these models relax the ACID properties in a specific manner to better model the processing requirements of a class of applications in terms of parallelism, consistency and serializability. For example, nested transactions [Mos85] allow subtransactions to fail independently of their respective parent transactions. This is appropriate for long transactions since the effects of failures is localized to parts of a transaction. Similarly, Sagas [GMS87], when compared to the traditional transaction model, relaxes the isolation and consistency properties thereby alleviating the problems caused by the all or nothing situation enforced by conventional transactions.

Several extensions to the conventional transaction model have been proposed in the literature to serve different application requirements. However, it is unlikely that this approach of rolling new variants of transactions as applications emerge will provide a realistic solution to the general problem. Furthermore, very little effort has been spent in addressing the implementation of these extensions as part of the database management system. Note also that it is not sufficient to support **an** extended transaction model in a DBMS as no single extended transaction model is likely to satisfy the transactional requirements of all applications. Clearly there is a need for: i) a DBMS to be configured with different transaction models as needed by the application, ii) a DBMS to support more than one transaction model at the same time, iii) configuring/selecting a transaction model by the user, and iv) to configure a DBMS with one or more transaction models.

Currently, a transaction model (traditional or otherwise) is typically hardwired into a DBMS. That is, the choice of the transaction model to be supported by a DBMS is made at the system implementation time thereby rendering it difficult to be changed. This is a severe limitation since the support of only one specific transaction model can only serve the requirements of a small class of applications. One solution to this problem is to come up with a transaction model that subsumes all other transaction models and satisfies the requirements of all classes of applications. This is unlikely to happen as evidenced by the current state-of-the-art and the growing list of requirements of applications such as workflow and activities that apply business rules across multiple databases in an enterprise.

Several alternative approaches to overcome this problem have been proposed by the research community and some of them have been incorporated into research prototypes although commercial DBMSs incorporate very few of these research results [Moh94].

- Object services architecture (OSA) is a software architecture consisting of a collection of independent (orthogonal) software services, all of which operate via a software backplane or message passing bus [Bla94]. TI's Open OODB prototype has taken this approach for supporting various services [WBT92, OOD93] but have not addressed the transaction model issues.

- Carnot [ASRS92] has taken the approach of providing a general specification facility that enables the formalization of most of the proposed transaction models that can be stated in terms of dependencies amongst significant events in different subtransactions. CTL (Computational Tree Logic) is used for the specification and an actor based implementation has been used for implementing task dependencies.

- ASSET [BDG$^+$94] identifies a set of primitives using which a number of extended transaction models can be realized. Implementation of the primitives has been sketched.

- ACTA [CR90] proposed a framework for specifying, analyzing, and synthesizing extended transaction models using dependencies.

- A proposal for supporting advanced transaction models by extending current transaction monitors' capability [Moh94].

As is evident, most of the research efforts have addressed newer transaction models and their semantics, specification of transaction models and reasoning over them, and utility of proposed transaction models. However, very little attention has been paid to supporting one or more transaction models, how to support newer transaction models in a system as they become available[1] and more importantly, the semantics and implementation details of supporting multiple transaction models.

In this paper, a pragmatic solution is proposed by adapting the active database paradigm for modifying the system behavior (as opposed to the application behavior) using sets of ECA rules. The basic idea is to allow the user/system designer to *build* or equivalently *emulate* the desired transaction behavior by using ECA (event-condition-action) rules to either: i) modify the behavior of a transaction model supported by the system or ii) support different transaction models (including the traditional one) by providing rule sets on primitive data structures.

In order to achieve the above goals it is necessary to examine extant transaction models to identify the requirements necessary to support them. Once these requirements are identified it is then possible to develop a framework that allows for the construction of these transaction models on an application by application basis. Examination of the different extended transaction models shows that it is necessary to provide a framework that supports: i) the specification of events, ii) the detection of events and iii) the execution of some action as a result of the occurrence of an event. To elaborate, consider the nested transaction model, if the top-level transaction aborts, then all of its children must be aborted. Therefore, it is necessary to detect the abort of a top-level transaction and then once this is detected, abort all of its children. As another example, when a child transaction commits, the locks should be inherited by the parent transaction. Thus it is necessary to detect the commit of a child transaction and once this is detected, give the locks to its corresponding parent.

In this work, we have started addressing the above issues. In this paper we argue how the active database functionality can be exploited at the systems level to address some of the required support outlined earlier. We propose several alternative approaches to supporting extended transaction models in databases in various ways. We believe that the approaches proposed in this paper will

[1]Incorporation of newer transaction models in already existing systems ideally should be achieved in a manner which does not require extensive changes to the existing system; the system should be extensible enough to facilitate the addition, deletion and modification of transaction models without resorting to extreme methods such as building layers of software to achieve this goal.

lead not only to the support of newer transactions models as they become available, but also to incorporate them into a DBMS along with existing ones relatively easily.

This paper is structured as follows. Section 2 outlines our approach based on the active database paradigm as well as presenting details of several alternative ways for supporting extended transaction models. Section 3 identifies the primitives which form the building blocks from which newer transaction models can be constructed. Section 4 provides an overview of Sentinel and illustrates our implementation strategy. In addition, it provides an example of how Sentinel can be used to model arbitrary transaction semantics. Section 6 includes related work, conclusions and future directions for research.

2 Active Database Approach to Flexible Transactions

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

In this paper, we propose to use active databases as a mechanism for specifying and enforcing the behavior of different transaction models. Active databases use ECA (event-condition-action) rules for providing active behavior. An ECA rule consists, primarily, of three components: an event, a condition, and an action. An event is an instantaneous, atomic (happens completely or not at all) occurrence. Conditions and actions correspond to side-effect free queries and transactions, respectively. Once an ECA rule is declared to the system, the active DBMS is responsible for detecting the event, evaluating the condition when the event occurs, and if the condition evaluates to true, executing the action. Therefore, extended transaction models can be specified and enforced by modeling them as ECA rules. For example, consider the nested transaction model where the commit of a top-level transaction can occur *only after* the termination of all of its subtransactions. In this case, the event component of the rule corresponds to the detection of a request to commit by a top-level transaction. When this event is detected, the condition checks whether the children of the transaction requesting to commit are still active, i.e., have not yet committed or aborted. If the condition evaluates to true (i.e., at least one child has not yet terminated), the action will postpone the commit of the top-level transaction until its children have terminated thereby enforcing the behavior of the nested transaction model.

We propose to use active databases as a uniform mechanism for providing the user with the ability to *construct* the required transaction semantics on an application by application basis. Specifically, we show how each transaction model can be translated into a set of ECA rules which can be activated or deactivated by users. For concreteness, we show how different transaction models can be specified in the context of an OODBMS, Sentinel. However, our approach is general enough to be applied by any active DBMS. Furthermore, we examine other techniques and provide a comparison between them and our approach based on adequacy, efficiency, flexibility and ease of use.

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

2.1 Alternatives for supporting extended transaction models

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

1. Provide a set of transaction primitives that allow users to define custom transaction semantics to match the needs of specific applications. This alternative assumes that the underlying DBMS supports some transaction model. This approach is taken in ASSET [BDG+94]. The user either directly uses these primitives or high-level notations (in the form of syntactic sugar) which are subsequently translated into the primitives supported by the system. This approach certainly enhances the functionality of the system and is a concrete approach towards supporting extended transaction models.

2. Provide a set of rules that the user can use from within applications to get the desired transaction semantics. This approach also assumes that the underlying DBMS supports some transaction model. However, the difference between this alternative and the previous one is the method by which the desired transaction semantics is obtained by the user. In the former, the user uses (or enables) the transaction primitives provided whereas in the latter the user uses the rule sets provided. For example, we assume that there is a set of rules for nested transactions that can be enabled by a command giving the user the semantics of nested transactions. Minimal user commands such as begin- and end-subtransaction are assumed. Similarly, another set of rules will provide the semantics of Sagas. Without any loss of generality we shall assume that rules are in the form of ECA rules, i.e., event, condition and action rules (along with coupling modes, event contexts, priority etc).

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

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

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

This approach is similar to the one taken in [US]. They introduce a flexible and adaptable tool kit approach for transaction management. This tool kit enables a database implementor or applications designer to assemble application-specific transaction types. Such transaction types can be constructed by selecting a meaningful subset from a starter set of basic constituents. This starter set provides, among other things, basic components for concurrency control, recovery, and transaction processing control.

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

We are aware that a number of issues need to be addressed in order to obtain solutions to approaches iii) and iv). In this paper, we will be concentrating on approach ii) which we believe is a good starting point that will lead to insights into how the rest of the approaches can be solved.

3 Primitives for Supporting Newer Transaction Models

In order to provide a framework that supports newer transaction models, it is first necessary to identify the set of *primitives* which form the building blocks from which various transaction models can be constructed. These primitives can be derived by examining the behavior of extant transaction models. In this section we examine transaction and dependency constraints, concurrency requirements, and access requirements of three transaction models, specifically, nested transactions, DOM transactions and sagas, aiming at identifying these primitives.

3.1 Nested Transactions

A nested transaction [Mos85] consists of a top-level transaction T and a set of component transactions C referred to as subtransactions. Each component transaction can in turn be a nested transaction. This model was proposed to overcome two main limitations of the traditional transaction model namely, limited parallelism and inflexible failure control. Subtransactions can be executed concurrently while ensuring execution atomicity; since nested transactions preserve serializability among their subtransactions, they can neither cooperate nor share data. Furthermore, the failure of a subtransaction does not necessarily cause the abort of the entire transaction.

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

In the nested transaction model, a child transaction must start after its parent transaction and terminate before it. In other words, a top-level transaction T cannot commit until all its children either commit or abort. If a top-level transaction T aborts then this will cause the abort of all its children. Hence, the commit of a subtransaction is conditionally subject to the commit or abort of its superiors. Even if a subtransaction commits, aborting one of its superiors will undo its effects. Updates of a subtransaction become permanent only when the enclosing top-level transaction commits. On the other hand, if a child fails, the parent is not required to abort. More specifically, failed portions of a transaction can be retried, compensated by attempting another alternative, or even ignored. This is basically application dependent.

The visibility rules of the nested transaction model are that a child transaction gets to see the latest version accessible to its parent rather than the original version. The children can view the partial results of their ancestors, the partial results of their committed siblings, plus any results from committed detached transactions. The delegation specification states that, at commit, the child transaction's objects are delegated to the parent transaction. This delegation makes the effects of committing child transactions selectively visible to the parent and the parent's other descendants. Furthermore, in nested transactions, a subtransaction can access without conflicts any object currently accessed by one of its ancestors. If a transaction requests a lock, the request can be granted only if all holders of conflicting locks (if any) are ancestors of the requesting transaction. When a transaction succeeds, all its locks are either inherited by its parent or released in case the transaction is top-level. When a transaction aborts, all its locks are discarded. If any of its superiors hold a lock on the same object, they continue to do so.

3.2 Sagas

Sagas [GMS87] have been proposed as a transaction model for long lived activities. A saga is a set of relatively independent (component) transactions T_1, T_2, ..., T_n which can interleave in any way with component transactions of other sagas. Component transactions within a saga execute in a predefined order which, in the simplest case, is either sequential or parallel (no order). Each component transaction T_i ($1 <= i < n$) is associated with a compensating transaction CT_i. A compensating transaction CT_i undoes, from a semantic point of view, any effects of T_i, but does not necessarily restore the database to the state that existed when T_i began executing. Both component and compensating transactions behave like atomic transactions in the sense that they have the ACID properties.

Component transactions can commit without waiting for any other component transactions or the saga to commit. However, a saga commits only if all its component transactions commit in the prescribed order. When a saga aborts, compensating transactions are executed in reverse order of commitment of the component transactions. A compensating transaction can commit only if its corresponding component transaction commits but the saga to which it belongs aborts. Due to their ACID properties, component transactions make their changes to objects effective in the database at their commitment times.

3.3 DOM Transaction Model

The DOM [BOH$^+$] transaction model consists of building blocks from which more complex transactions can be constructed. The DOM transaction model can behave primarily in three ways depending on the application requirements; it can behave as a conventional flat transaction model, like a transaction model that allows for closed nesting and the execution of triggered processes, or it can be used in its most powerful and flexible form by defining combinations of closed and open nestings.

In DOM, closed nested transactions are referred to as *toptransactions* which are basically transactions which make their results visible to the entire systems upon their commit. Toptransactions can be combined into multitransactions that have some global transaction semantics but allow for the visibility of partial results outside the multitransaction. Transactions within a DOM multitransaction can be executed either in parallel (which is the default execution semantics) or in some predefined order. When a multitranaction aborts, it causes the abort of its respective component transactions. If, however, a component transaction has already committed, a compensating transaction is initiated. Conversely, if a vital component transaction aborts it causes the abort of the multitransaction. If the component transaction is non-vital, then the execution of the multitransaction may continue.

3.4 Required Primitives

From the above it is possible to identify a set of primitives that can be used as a basis for building different transaction models. It is important to notice that the behavior of different types of transactions is determined at *key* points during the execution of a transaction. Let us consider the behavior of the nested transaction model when a top-level transaction aborts versus the behavior of the Sagas transaction model when the saga aborts. In the former, the abort of the top-level transaction causes the abort of all its respective children, while in the latter a sequence of compensating transactions will be executed in some prescribed order. Thus, it is necessary to support the *abort* primitive to model these two transaction models, however the semantics of the abort differs in these two models. This example clearly shows how active functionality can be used as a means to model different transaction models. It is necessary to treat the abort of a transaction as a primitive, in this case it will be an event, and once this event is detected, evaluate some condition and if it evaluates to true execute some action. If the required transaction model is the nested transaction model, once the system detects the abort of a transaction, it should check whether the transaction is a top level transaction. If the condition evaluates to true, i.e., the transaction is a top level transaction, the system should abort its respective children. However, if the required transaction model is Sagas, then once the system detects the abort of a transaction, it should check that the transaction that aborted is a component transaction, and if that condition evaluates to true, start executing the compensating transactions in reverse order to the commit of the component transactions.

Other examples of primitives that need to be supported are the *delegate* and *release* primitives. Consider the actions necessary to be performed when a child commits in the nested transaction model. First, the objects that were modified by the child transaction should be *delegated* or *transferred* to the respective parent who then becomes responsible for their commitment to the database. The parent will commit those delegated objects only if the parent commits. Second, the child must release all its locks upon commit and these locks are then inherited by the respective parent. Hence, the primitives *delegate* and *release* are required to model various transaction models or behaviors. We also introduce a new primitive that needs to be supported in order to model various transaction models. This primitive is the *suspend* primitive which causes the execution of a transaction to be temporarily suspended (until some event occurs which causes it to resume operation). A situation where this primitive is needed is in nested transactions, where the top level transaction needs to be suspended until its subtransactions terminate.

In this paper we claim that the following primitives are sufficient for constructing various transaction models and show how the nested transaction model and Sagas can be modeled using ECA rules which utilize these primitives in the event, condition and action components. The primitives are :

- t $\rightarrow$ start() : start execution of transaction t.
- t $\rightarrow$ commit() : commit the operations of transaction t.
- t $\rightarrow$ abort() : abort transaction t.
- t $\rightarrow$ suspend() : suspend execution of a running transaction t.
- t $\rightarrow$ delegate(s) : transfer objects from transaction t to transaction s.
- t $\rightarrow$ release(s) : release the locks held by a transaction t to transaction s.

3.5 Modeling Transactions using ECA rules

In this section we show how two transaction models namely, nested transactions and sagas can be modeled as a set of ECA rules. A high-level notation of ECA rules is used and the condition portion of the rules are simplified by making the events more specific. For example, instead of making the event the commit of a transaction, we make the events as the commit of a top level transaction or the commit of a child. Below, we show how the nested transaction model can be translated into ECA rules.

3.5.1 Modeling Nested Transactions using ECA rules

The set of ECA rules that model the semantics of nested transactions are:

- **On** Start of child
  Condition Parent has not started
  Action Reschedule start of child

- **On** Commit of top-level transaction
  Condition Children not terminated
  Action Reschedule commit

- **On** Abort of top-level transaction
  Condition True
  Action Abort all children

- **On** Commit of child
  Condition True
  Action Delegate all objects to parent

- **On** Abort of child
  Condition True
  Action Abort parent OR Retry child OR Compensate OR Ignore

- **On** Commit of top-level transaction
  Condition True
  Action Make permanent all updates of committed subtransactions

- **On** Commit of child
  Condition True
  Action Give locks to parent

- **On** Request of lock
  Condition Holders of conflicting locks are ancestors
  Action Grant lock

- **On** Abort of transaction
  Condition True
  Action Discard locks

3.5.2 Modeling Sagas using ECA rules

In this subsection we identify the set of rules which models the behavior of Sagas. These ECA rules are:

- **On** Commit of compensating transaction CT_i
  Condition Commit of T_i and Abort of saga
  Action Allow commit of CT_i

- **On** Commit of T_i
  Condition Commit of T_i does not violate predefined order
  Action Allow commit of T_i

- **On** Commit of T_i
  Condition True
  Action Make updates to object persistent

- **On** Abort of saga
  Condition True
  Action Execute compensating transactions in reverse order of commitment of component transactions

4 Implementation Using Sentinel

4.1 Overview of Sentinel

Sentinel [Cha91, CBM91, CM91, CM94, CG91, Sha92, AMC93, Bad93, CKAK94, CKTB94, Kri94, Tam94] 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 Open OODB class hierarchy was modified to include new class definitions which are necessary for supporting active capability. Figure 1 depicts the class hierarchy of Sentinel with respect to the Open OODB classes and the classes introduced, namely the *Reactive*, *Notifiable*, *Event*, *Rule* and *Event Detector* classes. Concurrency control and recovery for top-level transactions are provided by the Exodus storage manager.

In Sentinel, objects are classified into three categories: passive, reactive and notifiable. **Passive objects** are conventional objects which receive messages, perform some operations and then return results. They do not generate events. An object that needs to be monitored (by informing other objects of its state changes) cannot be passive. **Reactive objects**, on the other hand, are objects that need to be monitored (i.e., on which rules will be defined). A reactive object can declare any, possibly all, of its methods as an *event generator*. All methods declared as event generators constitute a reactive object's *event interface*. Once a method is declared as an event generator, its invocation will generate a primitive event. The primitive event can be generated either *before* or *after* the execution of the method depending on which *event modifier* was specified by the user. The event will be generated before execution and after execution if the user specifies the *begin* and *end* modifier, respectively. In addition, if the user specifies both modifiers then two primitive events will be generated, one before execution and one after execution of the respective method. To elaborate, let us consider a reactive object X whose method Y is declared as an event generator with the *end* modifier. Whenever object X invokes method Y and after Y is executed, a primitive event will be generated. Lastly, **Notifiable objects** are those objects that are capable of being informed of the events produced by reactive objects. Therefore, notifiable objects become aware of a reactive object's state changes and take appropriate measures (by evaluating conditions and executing actions) in response to those state changes. Notifiable objects subscribe to the primitive events generated by reactive objects. After the subscription, the reactive objects propagate their generated primitive events to the notifiable objects. Events and rules are examples of notifiable objects. Rules receive events from reactive objects, send them to their local event detector, and take appropriate actions. Event detectors receive events from reactive objects, store them along with their parameters, and use them to detect primitive and complex events. In the following paragraphs we briefly outline the implementation of the *Reactive*, *Notifiable*, *Event* and *Rule* classes. The reader is referred to [AMC93] for a detailed implementation of these classes.

The Reactive Class: The public interface of the Reactive class consists of methods by which objects acquire reactive capabilities. For an object to be reactive, i.e., have the ability to generate primitive events when methods in its event interface are invoked, it must be an instance of a class derived from the Reactive class[3]. Subclasses of the Reactive class will inherit several methods the most important of which is the *Subscribe* method. This method allows Notifiable objects to subscribe to the primitive events generated by instances of subclasses of the Reactive class. Once this subscription takes place, the notifiable object will be informed of the primitive events generated by the Reactive object. For example, if X is a Reactive object and Y is a Notifiable object, then

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

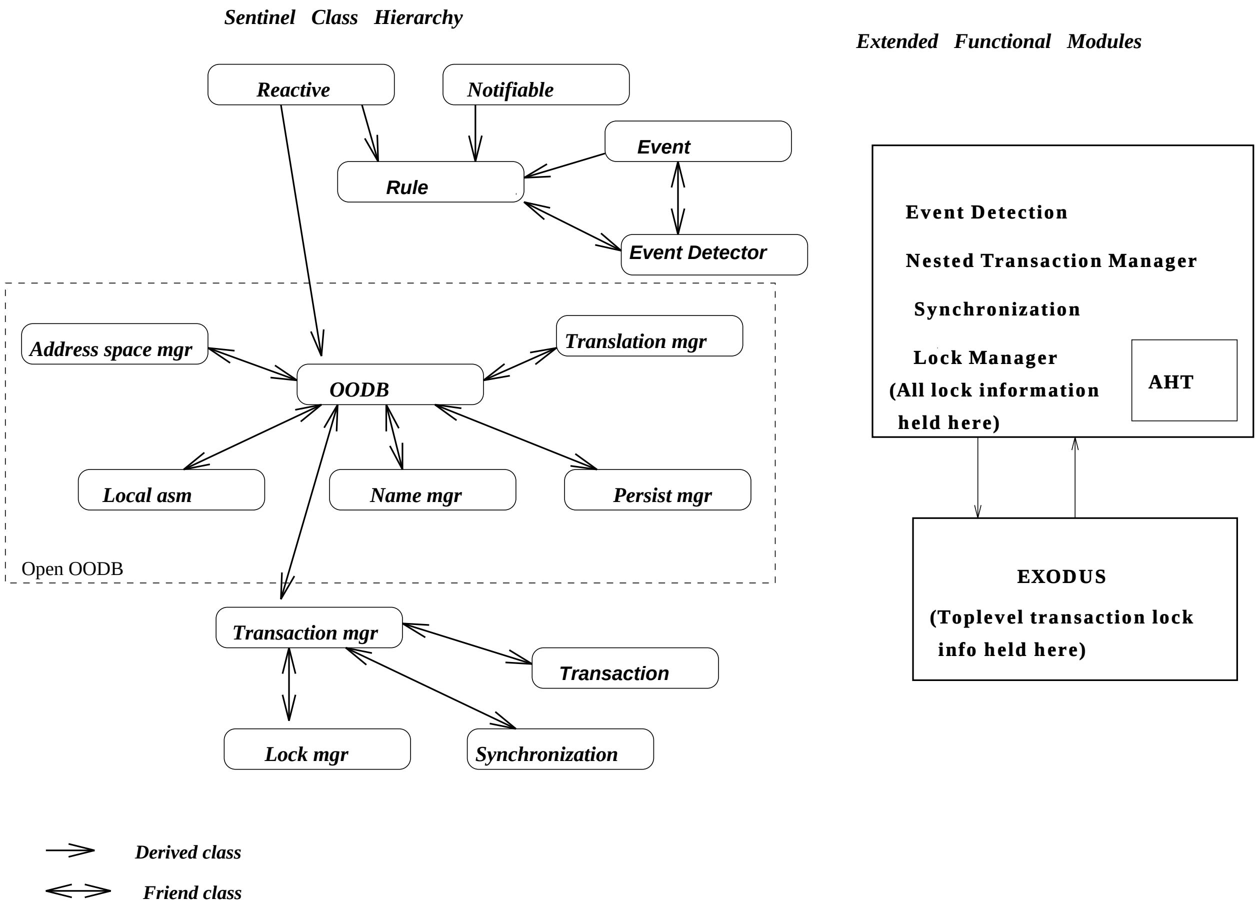

Figure 1: Sentinel class hierarchy

Y will be informed of the primitive events generated by X after the statement **X.Subscribe(Y)** is executed.

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

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

The Rule Class : The primary structure defining a rule is the event which triggers the rule, the condition which is evaluated when the rule is triggered, and the action which is executed when the rule is triggered. Therefore, creation of a rule object X is accomplished by executing the statement

```
class Transaction  {
   TID tid;
   TID parent_id;
   char status;
     .
     .
     .

   public:
      event begin && end start();                   /* event interface */
      event begin && end commit();                  /* event interface */
      event begin && end abort();                   /* event interface */
      event begin && end suspend();                 /* event interface */
      event begin && end delegate(TID transid);          /* event interface */
      event begin && end release(TID transid);           /* event interface */
};
```

Figure 2: The Transaction Class.

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

4.2 Adequacy of Sentinel for Supporting Transaction Models

As previously pointed out, the behavior of different transaction models is determined at *key* or *significant* points during the execution of a transaction. Therefore, in order to model these different transaction models it is necessary to *monitor* transactions to determine when these *key* events take place and subsequently react to them. Examples of key points are the start, commit, abort etc. of transactions. Therefore, to support different transaction models in Sentinel, all that is required is to make the Transaction class a subclass of the Reactive class and specify *all* the methods in the Transaction class to be event generators. It is necessary to specify all the methods of the Transaction class as event generators so that the user has the flexibility of customizing arbitrary transaction semantics in response to any action or combination of actions taken by a transaction. To elaborate, if the *commit* method is not specified as an event generator, then no transaction will generate a primitive event when it invokes the *commit* method, thereby rendering it impossible for the user to create rules which respond to the commit of a transaction.

Once the Transaction class is made a subclass of the Reactive class and all its methods declared as event generators, then any transaction object will generate events at significant points during its execution. Furthermore, if a rule (which is a notifiable object) subscribes to the events generated by a transaction object or a group of transaction objects, then it can react to those significant events in such a way that enforces the required transaction semantics.

Given the above, examination of the Sentinel architecture shows that it is adequate for supporting flexible transaction models. More specifically since the Reactive class is a superclass of the OODB class and the Transaction class is a friend class of the OODB class, then instances of the Transaction class are reactive objects. Therefore, all that is required is to declare the methods of the Transaction class as event generators. The Transaction class is depicted in Figure 2. Note that every method is declared as an event generator and that both event modifiers are used with these

```
Transaction T1, T2, T3, T4;

Event* commit    = new Primitive ("end Transaction::Commit()");          /* Event creation */
Event* abort  = new Primitive (" end Transaction::Abort() ");            /* Event creation */
Rule OrderDepedency(commit, True(), StartOtherTransactions(T2,T3,T4));    /* Rule creation */
Rule AbortDepedency(abort, True(), AbortOtherTwo());                     /* Rule creation */
T1.Subscribe(OrderDependency);    /* AbortDependecy rule subscribes to events generated by T1 */
T2.Subscribe(AbortDependency);    /* AbortDependecy rule subscribes to events generated by T2 */
T3.Subscribe(AbortDependency);    /* AbortDependecy rule subscribes to events generated by T3 */
T4.Subscribe(AbortDependency);   /* AbortDependecy rule subscribes to events generated by T4 */

T1->Start();
```

Figure 3: An Example of Modeling Transaction Semantics using ECA Rules.

methods. Therefore, the invocation of any of these methods will generate two events, specifically before and after its execution. In the following section, we give a concrete example of how Sentinel can be used to model arbitrary transaction semantics.

4.3 Example

In this subsection we show how an arbitrary execution of transactions can be enforced using ECA rules. More specifically, we illustrate how the following transaction sequence can be modeled. Assume there are four transactions T1, T2, T3, and T4 and that transactions T2, T3 and T4 can be executed in parallel provided T1 has already successfully committed, i.e., there is an order dependency between the executions of transaction T1 and transactions T2, T3 and T4. Furthermore, there is a commit dependency between transactions T2, T3 and T4, i.e., if any one of the transactions T2, T3 and T4 aborts, then the other two transactions must also abort.

To model the above transaction sequence it is necessary to monitor the significant events of transaction T1 and react to the commit of transaction T1 by executing transactions T2, T3 and T4 in parallel. Therefore, it is necessary to create a rule object which subscribes to the events generated by transaction object T1. This rule is triggered only when transaction T1 generates the commit event. Furthermore, we need to create another rule which monitors the events generated by transactions T2, T3 and T4. This rule is triggered when any one of these transactions T2, T3 or T4 generates the abort event. When this occurs, the rule should react by aborting the other two transactions. The code which accomplishes the above is depicted in Figure 3.

The code in Figure 3 creates four Transaction objects T1, T2, T3 and T4, two event objects **commit** and **abort**, as well as two rule objects **OrderDependency** and **AbortDependency**. The rule object **OrderDependency** takes as its parameters the event object **commit** and the two functions **True()** and **StartOtherTransactions(T2,T3,T4)**. Since this rule subscribes to transaction T1, it will be triggered after the execution of the method Commit by transaction T1; this is specified in the parameter of the event object **commit**. Once the rule is triggered, the condition function True() will be executed. This is a system defined function which always returns true, thus the action function StartOtherTransactions(T2,T3,T4) will be executed. The function **StartOtherTransactions** starts executing the transactions T2, T3 and T4 in parallel. Similarly, the rule object **AbortDependency** subscribes to the Transaction objects T2, T3 and T4. Therefore, all events generated by transactions T2, T3 and T4 will be propagated to rule object **AbortDependency**. This rule, however, is triggered only when it receives the **Abort** event, i.e., when any one of transactions T2, T3 and T4 abort. The rule then executes the condition function True() which always returns true and thus the action function AbortOtherTwo() is executed. Notice, that AbortOtherTwo does not take any parameters but will be able to determine which transaction has aborted and subsequently which are the other two transactions since the parameters of the event are automatically made available to the condition and action functions by the system. This example illustrates the ease and the flexibility for supporting arbitrary transaction semantics using Sentinel.

5 Conclusions

Before presenting our concluding remarks and future directions for research we give a brief overview of some related work.

5.1 Related Work

ASSET [BDG+94] is a system that provides a set of *transaction primitives* that allows users to define customized transaction semantics in applications. Transaction primitives are classified into basic and new primitives. Basic primitives are similar to those found in most transaction processing systems and are *initiate(f, args), begin(t), commit(t), wait(t), abort(t), self()* and *parent()*. The new primitives permit the construction of arbitrary transaction models and the realization of relaxed correctness criteria and are *delegate(t_i, t_j, ob-set), permit(t_i, t_j, ob-set, operations)* and *form-dependency(type, t_i, t_j)*. These transaction primitives are not expected to be directly used by the user; a high-level description of the required transaction model is specified by the user which is subsequently translated into code which uses these primitives. Below, we provide a brief description of these primitives.

Briefly, initiate(f, args) registers a new transaction that executes the function f with the arguments *args*. The primitives begin(t), commit(t) and abort(t) respectively start, commit and abort the transaction whose tid is t. The self() and parent() primitives each return a transaction tid; the former returns the tid of the executing transaction and the latter the tid of the executing transaction's parent. Waiting for a transaction t to complete is accomplished by using wait(t). The primitive delegate(t_i, t_j, ob-set) transfers the responsibility of operations performed on *ob-set* by transaction t_i to transaction t_j, i.e., these operations are committed only if transaction t_j commits (unless transaction t_j delegates them to another transaction). Cooperation amongst transactions is achieved by using the permit primitive; permit(t_i, t_j, ob-set, operations) means that transaction t_i permits transaction t_j to perform conflicting *operations* on objects in *ob-set* without conceptually creating a conflict edge in the serialization graph from t_i to t_j. The semantics of this operation allows t_j to execute *operations* on objects in *ob-set* without having to wait, only one transaction can perform an update operation at any given time and once a transaction t_i permits t_j to perform an operation on an object, t_j can permit other transactions to perform operations on that object. The last primitive form-dependency(type, t_i, t_j) establishes a dependency of the specified *type* between t_i and t_j, where *type* can be any type of dependency such as commit, abort and group commit.

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

The transaction dependency graph is a directed graph where the nodes represent transactions and the arcs represent the type of dependency between two nodes. For example, an arc of type commit from transaction t_i to transaction t_j denotes a commit dependency between the two transactions. For a detailed discussion of how the primitives and these data structures interact we refer the reader to [BDG+94].

5.2 Summary and Future Directions

In this paper, we have argued for the use of the active database paradigm at the systems level to support flexible transaction models. We proposed several alternative ways in which this can be accomplished and discussed their relative merits. We have analyzed a set of transaction models and arrived at a set of rules for supporting a couple of them at the conceptual level. We have shown how Sentinel architecture can be used to implement one of the alternatives proposed.

Our approach is different from ASSET in that our approach is based on the active database paradigm. In addition, we have proposed several approaches with different advantages and ease of design and implementation. Unlike the ASSET approach, we believe that our approach will provide benefits both for the designers of the DBMS and the users of the systems.

This paper only addresses a small portion of the problem described in this paper. Currently, we are investigating the alternatives iii) and iv) outlined at the beginning of the paper. Even for the approach ii), we are currently implementing it on Sentinel to better understand the interactions of 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.

[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*, pages 44–54, Minneapolis, Minnesota, May 1994.

[Bla94] Jose A. Blakeley. Open Object Database Management Systems. In *Proceedings, International Conference on Management of Data*, page 520, Minneapolis, Minnesota, May 1994.

[BOH$^+$] A. Buchmann, M. T. Ozsu, M. Hornick, D. Georgakopoulos, and F. Manola. *A Transaction Model for Active Distributed Object Systems.*

[CBM91] S. Chakravarthy and R. Blanco-Mora. Supporting very large production systems using active dbms abstraction. Technical Report UF-CIS TR-91-25, Database Systems R&D Center, CIS Department, University of Florida, E470-CSE, Gainesville, FL 32611, Sep. 1991.

[CG91] S. Chakravarthy and S. Garg. Extended relational algebra (era): for optimizing situations in active databases. Technical Report UF-CIS TR-91-24, Database Systems R&D Center, CIS Department, University of Florida, E470-CSE, Gainesville, FL 32611, Nov. 1991.

[Cha91] S. Chakravarthy. Active Database Management Systems: Requirements, State-Of-The-Art, and an Evaluation. In H. Kangassalo, editor, *Entity-Relationship Approach: The Core of Conceptual Modeling*, pages 461–473. Elsevier Science Publishers, North-Holland, 1991.

[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.

[CKTB94] S. Chakravarthy, V. Krishnaprasad, Z. Tamizuddin, and R. Badani. ECA Rule Integration into an OODBMS: Architecture and Implementation. Technical Report UF-CIS-TR-94-023, University of Florida, E470-CSE, Gainesville, FL 32611, Feb. 1994. (In ICDE-95, Taiwan, March 1995.).

[CM91] S. Chakravarthy and D. Mishra. An event specification language (snoop) for active databases and its detection. Technical Report UF-CIS TR-91-23, University of Florida, E470-CSE, Gainesville, FL 32611, Sep. 1991.

[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.

[CR90] P. K. Chrysanthis and K. Ramamtitham. Acta: A framework for specifying and reasoning about transaction structure and behavior. In *Proceedings, International Conference on Management of Data*, pages 194–203, 1990.

[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.

[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.

[Gra81] J. N. Gray. The transaction concept: Virtues and limitations. In *Proceedings, International Conference on Very Large Data Bases*, pages 144–154, September 1981.

[HR83] T. Haerder and A. Reuter. *Principles of Transaction-Oriented Database Recovery.* ACM Computing Surveys, 1983.

[Kri94] V. Krishnaprasad. Event Detection for Supporting Active Capability in an OODBMS: Semantics, Architecture, and Implementation. Master's thesis, Database Systems R&D Center, CIS Department, University of Florida, Gainesville, FL 32611, March 1994.

[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.

[Mos85] E. Moss. *Nested Transactions, an Approach to Reliable Distributed Computing.* The MIT Press, 1985.

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

[OV91] M. T. Ozsu and P. Valduriez. *Principles of Distributed Database Systems.* Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[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.

[Sha92] A. Sharma. On extensions to a passive dbms to support active and multi-media capabilities. Master's thesis, CIS Department, University of Florida, Gainesville, 1992.

[Tam94] Z. Tamizuddin. Rule Execution and Visualization in Active OODBMS. Master's thesis, Database Systems R&D Center, CIS Department, University of Florida, Gainesville, FL 32611, May 1994.

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

[WBT92] D. Wells, J. A. Blakeley, and C. W. Thompson. Architecture of an Open Object-Oriented Database Management System. *IEEE Computer*, 25(10):74–81, October 1992.