ASSET: A System for Supporting Extended Transactions*

A. Biliris, S. Dar, N. Gehani, H. V. Jagadish
AT&T Bell Labs
Murray Hill, NJ 07974
{biliris, dar, nhg, jag}@research.att.com

K. Ramamritham
University of Massachusetts
Amherst MA 01003
krithi@cs.umass.edu

Abstract

Extended transaction models in databases were motivated by the needs of complex applications such as CAD and software engineering. Transactions in such applications have diverse needs, for example, they may be long lived and they may need to cooperate. We describe ASSET, a system for supporting extended transactions. ASSET consists of a set of transaction primitives that allow users to define custom transaction semantics to match the needs of specific applications. We show how the transaction primitives can be used to specify a variety of transaction models, including nested transactions, split transactions, and sagas. Application-specific transaction models with relaxed correctness criteria, and computations involving workflows, can also be specified using the primitives. We describe the implementation of the ASSET primitives in the context of the Ode database.

1 Introduction

The atomic transaction model, in conjunction with serializability, can be limiting in advanced database applications that function in distributed, cooperative, and heterogeneous environments. For example, CAD transactions can be long lived, and large software systems construction may require cooperative transactions. Such applications have diverse needs. This has lead to the search for more flexible correctness requirements [20] as well as the introduction of new transaction models [12] that extend the traditional atomic transaction model.

However, little has been done thus far to make this research work available to a database user. At best, a specific extended transaction model may be provided by a database – the user has little flexibility in specifying which one, and no flexibility in changing its semantics. Given the large number of extended transaction models that have been proposed and the absence of a clear winner, database vendors today would hesitate to incorporate any one model into a product.

This paper reports on our attempts to address this limitation. Instead of providing users with a predefined transaction model, which may not be appropriate for their applications, we provide a flexible transaction facility, called ASSET, which is based on a set of transaction primitives. The primitives, presented in Section 2, are inspired by the ACTA transaction framework, a formal framework designed to specify, analyze and synthesize extended transaction models [8, 9].

These transaction primitives can be used to define customized transaction models suitable for specific applications. In particular we show in Section 3 how the transaction primitives can be used to specify nested transactions, split transactions, sagas, and other extended transaction models described in the literature. Even though we use the word transaction in this paper to refer to computations that have transaction-like properties, the transaction primitives also allow us to specify activities or workflows that are typically composed of transactions-like components. We use syntax based on O++, the database programming language of Ode [1], for illustrating the various transaction models. We assume that the reader is familiar with C++ [21].

Our primitives are general enough to be incorporated in any database system. We are currently implementing these primitives in the context of the Ode database [1] using the EOS [3] storage manager. Details of the implementation are discussed in Section 4.

2 Transaction Primitives and their Meanings

A database is a collection of persistent objects. Transactions invoke operations on objects. Operations execute atomically and in mutual exclusion. A transaction

Proceedings of ACM-SIGMOD 1994 International Conference on Management of Data, Minneapolis, Minnesota, May 1994, pages 44 - 54.

program may also manipulate volatile data, such as local variables. However, we do not permit volatile data to persist across transaction boundaries. For example, a volatile global variable may not be used in multiple transactions.

We now discuss transaction primitives for defining customized transaction models. It should be emphasized that we do not expect a user to use these primitives directly. Instead, we expect that these primitives will be used in the code generated by a compiler for a database programming language, or an application development interface, that provides *high-level support* for transactions. Development of the details of such a compiler is part of our future work.

The transaction primitives can be classified into two categories: (1) *basic* primitives, similar to those found in most transaction processing systems. (2) *new* primitives, which can be used in conjunction with the basic primitives for building transactions with custom semantics.

2.1 Basic Transaction Primitives

A transaction accesses and manipulates the objects in the database by invoking operations on the objects. A transaction that has invoked operations on an object but has not yet committed, is *responsible* for the uncommitted operations.

A transaction that has been initiated but has not begun execution is said to have been *initiated.* A transaction is said to be *running* if it is executing its code, i.e., it has begun executing but has not yet completed. A transaction is said to have *completed* if its code has been executed. A transaction has been *terminated* if it has been *committed* or *aborted.* A transaction is *active* if it has begun executing, and has not terminated yet (it may be running or completed).

The basic primitives in ASSET are

- `initiate`(f, *args*): register a new transaction that will execute the function pointed to by f with arguments *args*. If successful, `initiate` returns the transaction identifier (tid); otherwise, it returns the null tid . The transaction does *not* start executing; execution is started by calling `begin`.

- `begin`(t): start execution of the transaction whose tid is t. `begin` returns 1 if successful; otherwise, it returns 0. Similarly, `begin`(t_1, t_2, ..., t_n) starts execution of transactions t_1, t_2, ..., t_n.

- `commit`(t): commit the operations of transaction t. `commit` returns 1 if t commits or has already committed; otherwise, if t is aborted, `commit` returns 0.

  `commit` is a blocking primitive – if issued before the transaction has completed execution, it waits until the execution completes; it may also be delayed to satisfy dependency specifications (discussed later).

- `wait`(t): wait for transaction t to complete. `wait` returns 1 once the execution of the code for transaction t completes, or if t has already committed. `wait` returns 0 if the transaction has aborted by the time `wait` is invoked.

  A `commit` issued after a `wait` may still block or abort, but only if dependency requirements so dictate.

- `abort`(t): abort transaction t. `abort` returns 1 if the abortion of t is successful; otherwise, that is, if t has already committed, it returns 0.

- `self()`: returns the tid of the executing transaction.

- `parent()`: returns the tid of the parent transaction, that is the transaction that `initiate`d the transaction which invokes `parent`. For top-level transactions the null tid is returned.

When a transaction completes (that is, the function executed by it completes execution), the locks held by the transaction are not released and its changes are not made persistent. Instead, the transaction manager records the completion of the transaction. A transaction is committed by explicitly invoking `commit`.

In addition to the above, ASSET has primitives to query the status of transactions, for instance, to determine whether a transaction has aborted. We do not describe them here.

2.2 New Primitives for Specifying Custom Transaction Semantics

ASSET introduces three new primitives, `delegate`, `permit`, and `form_dependency` for specifying custom transaction semantics.

- `delegate`(t_i, t_j, *ob_set*): transaction t_i transfers to transaction t_j the responsibility for the operations performed by t_i on objects in the set *ob_set* [9]. These operations are committed if and only if t_j commits (assuming t_j does not delegate them to another transaction). That is, once t_i delegates an object *ob* to t_j, it will be as if t_j, not t_i, has performed the operations on *ob* for which t_i was responsible prior to the delegation. One implication is that a subsequent operation on *ob* performed by t_i can conflict with an operation previously performed by t_i.

  Similarly, `delegate`(t_i,t_j) transfers from t_i to t_j the responsibility for *all* the operations that t_i, is currently responsible for.

  The granularity of delegation can be lowered from objects (i.e., all the operations performed by the invoking transaction on the these objects) to specific operations. However, we do not consider this option further in this paper.

- `permit(`t_i`, `t_j`, `*ob_set*`, `*operations*`)`: t_i allows t_j to perform operations that conflict with t_i's operations, without, conceptually, creating a conflict edge in the serialization graph from t_i to t_j. This primitive has the following semantics:

  1. invoking transaction t_i allows t_j to execute the specified *operations* on objects in the set *ob_set* without forcing t_j to wait. Thus t_j can view objects accessed by t_i even before t_i commits or aborts.
  2. Only one transaction can perform an (update) operation at any given time. That is, atomicity and mutual exclusion continue to apply to the elementary operations.
  3. Once t_i has permitted t_j to perform an operation on some object, t_j has the right to permit further sharing with other transactions. In particular, if we execute
     `permit(`t_i`, `t_j`, `*ob_set*`, `*operations*`)`, and
     `permit(`t_j`, `t_k`, `*ob_set'*`, `*operations'*`)`:
     the effect is as if the command:
     `permit(`t_i`, `t_k`, `*ob_set* $\cap$ *ob_set'*`, `*operations* $\cap$ *operations'*`)`
     had also been executed.

  Three additional forms of `permit` are supported: `permit(`t_i`, `t_j`, `*operations*`)` allows t_j to execute conflicting *operations* on *any* object. `permit(`t_i`, `t_j`)` allows t_j to execute *any* (conflicting) operation on *any* object. `permit(`t_i`, `*ob_set*`, `*operations*`)` allows *any* transaction to perform the specified conflicting *operations* on objects in the set *ob_set*.

- `form_dependency(`*type*`, `t_i`, `t_j`)`: form a dependency of the specified *type* between t_i and t_j; while many *types* of dependency can be formed [8], three that occur more often are:

  – `CD` (commit dependency): If both commit, t_j cannot commit before t_i commits, but if t_i aborts, t_j may still commit.
  – `AD` (abort dependency): if t_i aborts, t_j must abort.
  – `GC` (group commit): either both t_i and t_j commit or neither commits.

  AD covers *CD*. That is, an abort dependency implies a commit dependency. Also, a group commit dependency involving a set of transactions can be specified using pairwise GC dependencies. We have chosen to provide three explicit primitives for ease of use.

One noteworthy design decision is the separation of the initiation and the beginning of a transaction. As we illustrate in Section 3, this separation allows us to delegate to or permit sharing with an initiated transaction before this transaction begins execution.

We assume that transactions are not malicious. The issue of "protection" of transactions, especially in the context of delegation and permission, is outside the scope of this paper.

3 Realizing Various Transaction Models and Relaxed Correctness Requirements

We now show how the ASSET transaction primitives can be used to specify different transaction models. As mentioned before, we do not expect a user to program with these primitives directly.

We refer the reader to [12, 9] for an in-depth discussion of different extended transaction models and to see how they help in the construction of complex database applications mentioned in Section 1.

3.1 (Extended) Transaction Models

We show how to implement a variety of transaction models using our primitives. To give a high-level description of the examples, we first show the transaction written in pseudo-code, based on the O++ language[2], a superset of C++. *Italic* font is used for pseudo-code inventions, while `constant-width` font is used to indicate actual code. We then show the implementation of the transaction model using our primitives.

3.1.1 Atomic Transactions

Atomic transactions combine the properties of serializability and failure atomicity. These properties ensure that concurrent transactions execute without any interference as though they executed in some serial order, and that either all or none of a transaction's operations are performed.
In O++, a standard (not nested) transaction is specified as follows:

```
trans {
    transaction-body
}
```

The O++ compiler takes the transaction body and encapsulates it into a function `f` defined as

```
void f() {
      transaction-body
}
```

The compiler will also generate code to initiate, begin, and commit a transaction t executing the function f:

```
tid t;  // transaction tid
if  ((t = initiate(f)) != NULL) {
    if (begin(t)) {
        commit(t);
    }
}
```

Error checks on the return values of `initiate` and `begin` can be used to execute error-handling code, if desired. To minimize clutter, we omit these checks in the examples that follow.

3.1.2 Distributed Transactions

A distributed transaction consists of one or more component transactions executed in parallel. The component transactions can only commit as a group, resulting in the commitment of the distributed transaction. Consider the following distributed transaction (`||` denotes parallel execution):

```
trans { f1() } || trans { f2() }
        || ... || trans { fn() }
```

The above distributed transaction is translated as follows:

```
tid t1, t2, ..., tn;
t1 = initiate(f1); t2 = initiate(f2);
         ...; tn = initiate(fn);
form_dependency(GC, t1, t2);
         ...; form_dependency(GC, t1, tn);
begin(t1); begin(t2); ...; begin(tn);
commit(t1); commit(t2); ...; commit(tn);
```

The group commit/abort semantics are accomplished by the specification of group dependency. Consequently, it is sufficient to commit or abort only one of the transactions. In this example `commit(t1)` actually accomplishes the group commit of all the transactions in the group, and the remaining `commit` invocations simply return `1` to indicate that their respective argument transactions have already committed. Similarly, if the group commit attempted by `commit(t1)` does not succeed, all the transactions abort. Later `commit` invocations simply return `0`.

3.1.3 Contingent Transactions

A contingent transaction consists of two or more component transactions. At most one of the component transactions of a contingent transaction commits; the component transactions are executed in the order specified.
Consider the following contingent transaction:

```
trans {f1()} else trans {f2()}
         else ... else trans { fn() }
```

The above contingent transaction is translated as follows:

```
tid t1, t2, ... tn;
t1 = initiate(f1);
begin(t1);
if (commit(t1));
else {
    t2 = initiate(f2);
    begin(t2);
    if (commit(t2));
    else {
     ....
        else
        tn = initiate(fn);
        begin(tn);
        commit(tn);
} }  .... }
```

3.1.4 Nested Transactions

A nested transaction (*cf.* [18]) consists of a *root* (or *parent*) transaction and *nested* component transactions called subtransactions. The subtransactions can themselves be nested transactions. Subtransactions execute atomically with respect to their siblings. They are failure atomic with respect to their parent. That is, they can abort without causing the whole transaction to abort.

A subtransaction can potentially access any object that is currently accessed by one of its ancestor transactions without forming a conflict. `abort` semantics for both root transactions and subtransactions are similar to `abort` semantics in atomic transactions. However, `commit` semantics are different for root transactions and subtransactions. When a subtransaction commits, the objects modified by it are made accessible to its parent transaction. However, the effects on the objects are made permanent only upon the commit of the topmost root transaction.

We illustrate the translation of nested transactions via an example. This involves a two-level nested transaction used for making a trip arrangement:

```
tid t;
t = trans {
    trans { make_airline_reservation(); }
    trans { make_hotel_reservation(); }
}
```

If an airline ticket cannot be purchased, then the trip is canceled. If a hotel reservation cannot be made then too the trip is canceled. The effects of the airline reservation transaction must be undone in that case.

The above nested transaction is translated as follows.

```
tid t;
t = initiate(trip);
```

```
begin(t);
commit(t);
```

where function trip is synthesized as follows:

```
void trip()
{
   tid t1;
   t1 = initiate(make_airline_reservation);
   permit(self(), t1);
   begin(t1);
   if (!wait(t1))
       abort(self());
   delegate(t1, self());
   commit(t1);

   tid t2;
   t2 = initiate(make_hotel_reservation);
   permit(self(), t2);
   begin(t2);
   if (!wait(t2))
       abort(self());
   delegate(t2, self());
   commit(t2);
}
```

This is the first example illustrating the use of the `permit` and `delegate` primitives. We do not show the code for functions `make_airline_reservation` and `make_hotel_reservation`, executed by subtransactions `t1` and `t2` respectively. We assume that both subtransactions abort if they are unsuccessful in making the respective reservations. If they succeed, they delegate all the updates they have made to `t`, the top-level transaction. Otherwise `t` is aborted, in which case the updates made thus far, if any, are discarded. Note that since a reservation transaction delegates all its operations and does nothing afterwards, it does not actually matter whether this transaction is committed or aborted subsequent to the delegation.

3.1.5 Split and Join Transactions

In the split transaction model [19] a transaction, say t_a, can split into two transactions, t_a and t_b. At the time of the split, operations invoked by t_a on objects in a set `ob_set` (up to the split point) are delegated to t_b. t_a and t_b can commit or abort independently. Conversely, two transactions, say t_a and t_b, can *join* to form one transaction t_a.

Consider the following simple example of a transaction s that is split off from some transaction t. s subsequently joins the transaction t. The code for s is embedded in the function f. The transaction split, i.e.,

```
trans t {
    s = split trans X { f(); };
}
```

where X is the set of objects to be delegated to the split transaction, is translated as follows:

```
s = initiate(f);
delegate(parent(s), s, X);
begin(s);
```

The transaction join, i.e.,

```
join(s, t);
```

is translated as follows:

```
wait(s);
delegate(s, t);
```

3.1.6 Sagas

Sagas [13] 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 that can interleave in any way with component transactions of other sagas. Component transactions within a saga execute, in the simplest case, in a predefined sequential order.

Each component transaction t_i $(1 \leq i < n)$ 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. However, their behavior is constrained by certain dependencies. For example, a compensating transaction commits if only if its corresponding component transaction commits but the saga of which it is a part aborts.

Component transactions can commit, making their changes to objects effective in the database, without waiting for any other component transactions or the saga to commit. Thus, isolation is limited to the component transaction level and sagas may view the partial results of other sagas.

A saga commits, i.e., successfully terminates, if all its component transactions commit in the prescribed order. Under sequential execution, the correct execution of a committed saga is:

$$t_1 \, t_2 \cdots \, t_n$$

where t_1, ..., t_n are components of the saga.

A saga is not failure atomic but it cannot execute partially either. When a saga aborts, it has to compensate for the committed components by executing their corresponding compensating transactions. Compensating transactions are executed in the reverse order of commitment of the component transactions. Thus, in the sequential case, the correct execution of an aborted saga after the commitment of its k^{th} component transaction, t_k $(1 \leq k < n)$, is:

$$t_1\ t_2\ \cdots\ t_k\ ct_k\ ct_{k-1}\ \cdots\ ct_1$$

where ct_1 is a compensating transaction for t_1, ct_2 for t_2, etc. The commitment of t_n implies the commitment of the whole saga and hence t_n is not associated with a compensating transaction ct_n.

Consider the following saga

```
saga {
   trans { f1()}
     compensating trans { cf1()}
   trans { f2()}
     compensating trans { cf2()}
    ...
   trans { fn_1()}
     compensating trans { cfn_1()}
   trans { fn()}
}
```

The above saga is translated as

```
tid t1, t2, ..., tn_1, tn;
int i = 0;
{
    t1 = initiate(f1); begin(t1);
        if (!commit(t1)) break; i++;
    t2 = initiate(f2); begin(t2);
        if (!commit(t2)) break; i++;
     ...
    tn = initiate(fn); begin(tn);
        if (!commit(tn)) break; i++;
}
tid ct1, ct2, ..., ctn_1
switch (i) {
case n-1:
    do
        ctn_1 = initiate(cfn_1);
        begin(ctn_1);
    while (!commit(ctn_1));
 // a compensating transaction must be
 // retried until it finally commits.
 ...
case 2:
    do
        ct = initiate(cf2); begin(ct2);
    while (!commit(ct2));
case 1:
    do
        ct = initiate(cf1); begin(ct1);
    while (!commit(ct1));
default:
    ;
}
```

3.2 Relaxed Correctness Requirements

In this section we discuss three examples of cooperative behavior among transactions. These satisfy correctness requirements that are less stringent than serializability.

3.2.1 Cooperating Transactions

Assume that two transactions t_i and t_j want to work cooperatively on an object (read/write the same object).

For example, t_i allows t_j to perform conflicting operations by executing

```
form_dependency(CD, t_i, t_j)
permit(t_i, t_j, ob, op);
```

The CD requirement imposed by the first statement ensures that t_j cannot commit before t_i terminates. With the second statement, t_i allows t_j to perform operation *op* on object *ob* concurrent with t_i's accesses to *ob*.

Similarly, t_j can allow t_i to perform conflicting operations

```
permit(t_j, t_i, ob, op);
```

This "ping-ponging" of `permit`s allows the two transactions to cooperate with each other. Controlled forms of dependence may thus be introduced in return for relaxed concurrency semantics, in a manner that produces predictable interactions in a particular application scenario. In this example, once t_j permits t_i to perform conflicting operations, another CD could be established between t_j and t_i if we desire that the two cooperating transactions must both commit or neither. Such interactions would occur, for example, in cooperative design environments wherein changes to the (design) object being shared will be committed only if the final state of the object is considered to be acceptable in the eyes of the cooperating designers.

3.2.2 Cursor Stability

Commercial database systems provide for relaxed degrees of consistency. A popular choice is cursor stability, which allows writes by a transaction t_j to follow reads of an uncommitted (reading) transaction t_i on an object *ob* that t_i has finished reading. This results in non-repeatable reads by t_i.

In terms of our primitives, before moving the cursor from one record to the next within a relation, the reading transaction t_i executes

```
permit(t_i, record, write)
```

This permission allows *any* transaction to write the specified *record* without waiting for t_i to commit. No dependencies are formed, so that t_i and t_j may commit in any order.

3.2.3 Workflows

Workflows are long-lived activities with transaction-like components having inter-related dependencies [22, 11]. Here we show workflows can be captured using our primitives. Just as we had higher-level language constructs corresponding to each of the transaction models discussed earlier, it is possible to design a language to specify workflows. These would then be translated into the code given here.

Consider, as an example, person X who is traveling from New York to Los Angeles to attend a conference from June 11, 1994 to June 14, 1994. X wants to leave New York on June 11 and leave Los Angeles on June 14 and X must stay at hotel Equator, the site of the conference. X prefers to fly on Delta, United, or American in that order. X will not travel on any other airline. The car must be rented from Avis or National with which X's company has corporate accounts.

The workflow activity involves making flight and hotel reservations, and optionally a car rental reservation. If no flight or hotel is available, the whole trip is canceled. If a car cannot be rented, the trip can still proceed since X can take public transportation. The program for the workflow activity can be found in the appendix.

4 Implementation

This section describes the data structures and algorithms used to support the transaction primitives in a modified version of the EOS storage manager [3]. We focus our discussion here on one mode of operation in which the application operates directly on the objects in a shared cache without first copying the object to its private address space.

4.1 Data Structures

Data structures related to the implementation of the transaction primitives are as follows:

- The transaction descriptor (TD). It contains the tid of a transaction t and its parent (if any), the transaction status, and a list of t's lock requests. The transaction descriptors are placed in a chained hash table based on the transaction tid. TD_i denotes the descriptor of the transaction t_i.

- The object descriptor (OD). It provides information about a locked object and contains the id of the object, a list of granted locks requests, a list of pending lock requests for the object, and a list of permissions so that transactions can perform conflicting operations on the object, see Figure 1. Each object in the cache points to its own descriptor so no searching is needed when a lock on the object is requested. OD_{ob} denotes the descriptor of the object ob.

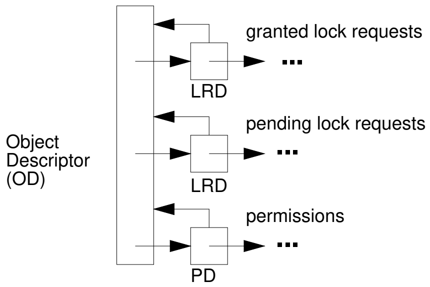

Figure 1: The object descriptor.

- The lock request descriptor (LRD). It describes a granted or pending lock request on a particular object ob by some transaction t_i. It includes a pointer to the TD_i, a pointer to the OD_{ob}, the lock mode of the request (read, write, none), the status of request (granted, pending, or upgrading), and a pointer to the next lock request of t_i.

- The permit descriptor (PD). Each OD of an object ob points to a (possibly empty) list of PDs. A PD contains a back pointer to an OD_{ob} and a triple (t_i, t_j, op) indicating that even if ob is currently locked by t_i in a mode that normally conflicts with op, t_j can still perform op on object ob as far as t_i is concerned. PDs are doubly hashed on the tid of the two transactions involved so that permissions given by or given to a transaction can be located efficiently.

- The transaction dependencies graph. This is a directed graph where the nodes represent transactions and an edge from t_i to t_j labeled with $type$ represents a dependency ($type$, t_i, t_j). The graph is stored in structures that include pointers to the TD of the transactions involved in the dependency as well as the type of dependency. These structures are doubly hashed on the tid of the two transactions involved so that dependencies emanating from or incoming to a transaction can be located efficiently.

Latches control simultaneous access to data and control structures in the shared memory region and ensure atomicity of read/write operations on the cached objects. There are two modes in which an item may be latched: shared (S) and exclusive (X). Before a shared item is accessed the latch associated with the item must be obtained in the appropriate mode. The latch is released immediately after the access completes.

Latches in EOS are implemented by an atomic test-and-set operation. If a process cannot (test-and-) set a latch it "spins" on it (perhaps with some time-varying delay) until the latch is unset. Each latch, in addition to the value that can be set or unset atomically, contains an S-counter indicating the number of processes holding the latch in S mode and an X-bit indicating whether a process is waiting to get the latch in X mode. The X-bit blocks new readers from setting the latch, thus preventing starvation of update transactions.

4.2 Implementation of Transaction Primitives

• `initiate`(f,$args$):
If resources are available, generate a tid and a TD for this transaction with the status set to *initiated*. If no resources are available (e.g., the number of transactions exceed a predetermined number) return an error code. Register the memory address of the function f that is to be executed by this transaction. Also record the tid of the parent transaction (null if this is a top level transaction).

• `begin`(t):
When the transaction begins executing its status, in its TD, is modified to *running*.

• `read-lock`(t_i, ob), `write-lock`(t_i, ob):

1. Scan the list of lock requests LRD pointed by OD_{ob}. For each granted lock gl recorded in an LRD do the following.
   (a) If this gl is a lock acquired by t_i, gl is not suspended, and gl "covers" the requested lock, return success.
   (b) If gl is held by t_j and conflicts with the requested lock, scan the list of permissions (PD) pointed to by OD_{ob}. If t_j permits t_i on ob, *suspend gl*. If no permission is given to t_i to place the lock, t_i blocks and retries later starting at step 1.
2. t_i can now lock ob.
   (a) If there is no LRD pertaining to t_i, create one indicating that t_i has a lock on ob and insert it in the list of granted locks pointed to by OD_{ob} and TD_i.
   (b) If there is already a LRD pertaining to t_i, change the lock mode, or remove suspension status, as the case may be.
3. Return success.

• `read`(t_i, ob):

1. read-lock(t_i, ob), if not already locked.
2. Get an S-latch on ob.
3. Perform the read operation.
4. Release the S-latch on ob.

• `write`(t_i, ob):

1. write-lock(t_i, ob), if not already locked.
2. Get an X-latch ob.
3. Write the before image of ob to the log.
4. Perform the write operation.
5. Write the after image of ob to the log.
6. Release the X-latch on ob.

To simplify the presentation, we omit the latch/unlatch steps from the following code.

• `form_dependency`($type$, t_i, t_j):
Insert a new edge in the dependency graph. Before this, a check is performed to prevent certain dependency cycles.

• `permit`(t_i, t_j, ob, op):
Make a permit descriptor PD that includes the (t_i, t_j, op) triple and insert it to the list of PDs pointed by OD_{ob}, and the PD graph. If t_j is null, the permission is given to all transactions. Also, if op is null, the permission is given for all operations on ob.
• `permit`(t_i, t_j, op):
For each object ob that t_i accessed or has permission to access perform `permit`(t_i, t_j, ob, op). These objects can be found by traversing the LRD list and the PDs of permissions given by t_i.
• `permit`(t_i, t_j):
invoke `permit`(t_i, t_j, $null$).
• `permit`(t_i, ob, op):
invoke `permit`(t_i, $null$, ob, op).

• `delegate`(t_i, t_j, ob_set):
For each ob in the ob_set, do the following: (a) remove the LRD pertaining to t_i on ob from the TD_i's list and insert it in the list of LRDs pointed by TD_j; (b) change any PD of the form (t_i, t_k, op) to (t_j, t_k, op).
• `delegate`(t_i, t_j):
Traverse the LRD list pointed to by TD_i and remove each LRD from the t_i's list and insert it in the LRD list pointed by TD_j. Update all PDs of permissions given by t_i to be permissions given by t_j.

• `commit`(t_i) :

1. Check the transaction's status in TD_i. If it is *committed* return success. If it is *aborted* return failure. If it is *aborting*, perform the steps of the `abort`(t_i) algorithm described below. Otherwise, change the status of t_i to *committing* and execute the following steps.
2. Scan the list of dependencies emanating from t_i and for each such dependency d of t_i on some transaction t_j do the following.

(a) If d is an abort dependency, t_i cannot commit because if t_j aborts, t_i must abort too. t_i blocks and retries later starting at step 1.

(b) If d is a commit dependency, t_i can only commit after t_j terminates (commits or aborts). t_i blocks and retries later starting at step 1.

(c) d is a group commit dependency.

    i. Leave a mark in this dependency edge indicating that t_i is waiting for t_j to commit.

    ii. If t_j has not left a mark on this dependency edge that waits for t_i to commit, invoke `commit(`t_j`)`.

    iii. If t_j has left a mark on this dependency that waits for t_i to commit, add t_j to the list of transactions that can commit as a group with t_i.

3. If there is at least one group commit dependency unresolved, t_i blocks and retries later starting at step 1. If there is a resolved group commit dependency involving t_i, the steps below are simultaneously executed for all the transactions in the group.

4. At this point t_i does not depend on any other transaction. Place a commit record in the log. Change the status of t_i to *committed*.

5. Scan the list of dependencies pertaining t_i and remove each such edge from the dependency graph. This will remove all dependencies of other transactions on t_i.

6. Release all locks held by t_i by traversing the *LRD* lists pointed to by TD_i, and remove permissions given by and given to t_i.

7. Return success.

- `abort(`t_i`)` :

1. Check the transaction's status recorded in TD_i. If it is *committed* return failure. If it is *aborted* return success. Otherwise, change the status of t_i to *aborting* and execute the following steps.

2. Scan the log and for each update performed by t_i on object *ob*, install the before image of *ob* with respect to *this* update. (This implies that subsequent updates done by "cooperating transactions" will also be lost.)

3. Release all locks held by t_i.

4. Scan the list of dependencies incoming to t_i and for each such dependency d of some transaction t_j on t_i do the following.

(a) If d is an abort or group commit dependency, t_j must abort. Mark t_j in its *TD* structure as *aborting*.

(b) If d is a commit dependency, just remove this dependency.

5. Remove the remaining dependencies pertaining to t_i from the dependency graph.

6. At this point no other transaction depends on t_i. Change the status of t_i to *aborted* and return success.

5 Conclusions and Future Work

In this paper we have developed a flexible transaction facility called ASSET. ASSET allows the specification of arbitrary transaction models and provides support for programming transactions that have relaxed correctness requirements. Our goal is to facilitate the construction of transactions that cooperate and interact in application-specific ways. Through several examples, we showed how three novel transaction primitives, namely, **`delegate`**, **`permit`** and **`form_dependency`**, allow the construction of arbitrary transaction models and the realization of relaxed correctness notions.

With regard to related work, Argus [17] represents one of the earliest efforts at providing linguistic supports for transactions. Argus, however, supported only nested transactions; each operation invocation was considered to be a (sub)transaction. More recently, efforts have been underway in several places to provide support for transactional activities. Among these, work similar in spirit to ours is in progress as part of the InterBase Project at Purdue University [7], the DOM project at GTE Laboratories[14] and in the context of the Contract model [22].

Flex, the model used in the InterBase Project, is based on the nested transaction model but allows relaxations of the atomicity and isolation requirements. Specifically, it has primitives to commit one of many transaction components and to execute multiple transaction components in parallel. Also, dependencies between transaction components can be specified. These contribute to the flexibility of transaction specifications. ASSET's flexibility, on the other hand, derives from the set of primitives designed explicitly to allow for the cooperation of transactions through object sharing and delegation. These primitives allow users specify the transaction behaviors just mentioned. As in Flex, dependencies between transaction components can be specified in ASSET.

The Transaction Specification and Management Environment (TSME) of the DOM project provides an ACTA-like specification language for users to express the properties of extended transactions. While DOM also allows the specifications of dependencies, unlike DOM, ASSET's primitives are *at the programming language level* but are based on ACTA building blocks.

In the Contract model, a set of steps define individual transactions and a script is provided to control the execution of these transactions. Even though the functions associated with transactions in our work may appear to be similar in concept to steps, transaction management primitives can be invoked from within our functions. Contract scripts introduce their own control flow syntax, while our approach uses the control flow syntax of the host language (e.g., O++) and only introduces a small set of transaction management primitives. As with contracts, ASSET allows explicit dependencies between transactions to be expressed using the `form_dependency` primitive.

One of our future goals is to develop further the underlying transaction management system of ASSET to exploit the concurrency semantics inherent in objects [10]. This will take advantage of the semantics of objects in an object-oriented database, by considering the compatibility of class specific operations (methods). We believe that many operations in an object-oriented database may commute. For example, operations to increase an existing employee's salary and to add a new employee to a department commute. Also, operations on an object commute provided they operate on different parts of the object. For example, operations that update an employee's salary and change the employee's department commute. Hence, we would like to capitalize on the semantics of objects in an object-oriented database, by taking into account the compatibility of class specific operations (methods). Concepts and mechanisms from Multi-level transactions [23] will come into play when we consider operation semantics.

Acknowledgments

We appreciate the feedback given to us by Thimios Panagos, Naser Barghouti, Joan Peckham, Marvin Solomon and Mike Zwilling.

References

[1] Agrawal R., and N. Gehani. Ode (Object Database and Environment): The Language and the Data Model. In *Proc. of the ACM SIGMOD Int'l Conference on Management of Data*, pages 35–45, 1989.

[2] Biliris A., N. Gehani, D. Lieuwen, E. Panagos, and T. Roycraft. Ode 2.0 User's Manual. AT&T Bell Laboratories, 1993.

[3] Biliris A., and E. Panagos. EOS: An Extensible Object Store. In this Proceedings.

[4] Badrinath, B. R. and K. Ramamritham. Semantics-Based Concurrency Control: Beyond Commutativity. *ACM Transactions on Database Systems*, Vol. 17, No. 1, March 1992, pp. 163-199.

[5] Beeri, C., H. J. Schek, and G. Weikum. Multilevel Transaction Management, Theoretical art or practical need?. *First Int. Conference on Extended Database technology*, Venice, 1988, Springer LNCS 303.

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

[7] Bukhres, O., A. Elmagarmid, and E. Kuhn. Implementation of the Flex Transaction Model. *Bulletin of the IEEE Technical Committee on Data Engineering*, 16(2):28–32, June 1993.

[8] Chrysanthis P. K. and K. Ramamritham. A Formalism for Extended Transaction Models. In *Proc. of the seventeenth Int'l Conference on Very Large Databases*, September 1991.

[9] Chrysanthis, P. K. and K. Ramamritham. Synthesis of Extended Transaction Models using ACTA. *ACM Transactions on Database Systems* (to appear).

[10] Chrysanthis P. K., S. Raghuram, and K. Ramamritham. Extracting Concurrency from Objects: A Methodology In *Proc. of the ACM SIGMOD Int'l Conference on Management of Data*, pages 108–117, 1991.

[11] Dayal U., M. Hsu, and R. Ladin. Organizing Long-Running Activities with Triggers and Transactions. In *Proc. of the ACM SIGMOD Int'l Conference on Management of Data*, pages 204–214, Atlantic City, May 1990.

[12] Elmagarmid, A. K., editor, *Database Transaction Models for Advanced Applications*. Morgan Kaufmann, 1991.

[13] Garcia-Molina H. and Kenneth Salem. SAGAS. In *Proc. of the ACM SIGMOD Int'l Conference on Management of Data*, pages 249–259, May 1987.

[14] Georgakopoupos, D., M. Hornick, P. Krychniak, and F. Manola, Specification and Management of Extended Transactions in a Programmable Transaction Environment. *IEEE Conference on Data Engineering*, Feb 1994.

[15] Korth H. F., W. Kim, and F. Bancilhon. On Long-Duration CAD Transactions. *Information Sciences*, 46(1-2):73–107, October-November 1988.

[16] Korth H. F., E. Levy, and A. Silberschatz. Compensating Transactions: A New Recovery Paradigm. In *Proc. of the Sixteenth Int'l Conference on Very Large Databases*, pages 95–106, Brisbane, Australia, August 1990.

[17] Liskov B. and R. Scheifler. Guardians and Actions: Linguistic support for robust distributed programs. *ACM Transactions on Programming Languages and Systems* Vol 8, No.4, December 1983, pp 484–502.

[18] Moss J. E. B. *Nested Transactions: An approach to reliable distributed computing.* PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, April 1981.

[19] Pu C., G. Kaiser, and N. Hutchinson. Split-Transactions for Open-Ended activities. In *Proc. of the Fourteenth Int'l Conference on Very Large Databases*, pages 26–37, Los Angeles, California, September 1988.

[20] Ramamritham, K. and P. Chrysanthis. In Search of Acceptability Criteria: Database Consistency Requirements and Transaction Correctness Properties. *Distributed Object Management*, Ozsu,Dayal, and Valduriez Ed., Morgan Kaufmann, 1993.

[21] Stroustrup, B. and M. A. Ellis. *The Annotated C++ Reference Manual*, Addison-Wesley, 1990.

[22] Wächter, H. and A. Reuter. The ConTract Model. in [12].

[23] Weikum. G. Principles and Realization Strategies of Multi-Level Transaction Management. *ACM TODS*, 16(1):132 – 180, March 1991.

Appendix – The Workflow Program

```
//Suppose that the following functions have been defined that make (or cancel) the appropriate
// reservations. The last two functions respectively compensate for the first two functions.

  void flight_reservation(Airline air, Date d1, Date d2);
  void hotel_reservation(Hotel h, Date d1, Date d2);
  void car_reservation(CarRent c, Date d1, Date d2);
  void cancel_flight_reservation(Airline air, Date d1, Date d2);
  void cancel_hotel_reservation(Hotel, Date d1, Date d2);

//Using these functions, the desired workflow can be defined as follows:

void exclusive_car_reservation(CarRent car, Date d1, Date d2, tid t);
{ car_reservation(car, d1, d2);
   if (wait(self())) abort(t);}

int X_conference()
{ tid t1, t2, t3, t4, t5, t6;
   Airline* air;
   t1 = initiate(flight_reservation, "Delta", "6/11/1994", "6/14/94");
   begin (t1);
   if (!commit(t1)) {
      t2 = initiate(flight_reservation, "United",  "6/11/1994", "6/14/94");
      begin (t2);
      if (!commit(t2)) {
         t3 = initiate(flight_reservation, "American",  "6/11/1994", "6/14/94");
         begin (t3);
         if (!commit(t3)) return 0; // Activity Failed
         else air = "American";}
      else air = "United";}
   else air = "Delta";
   // Flight reservation has been made at this point

   t4 = initiate(hotel_reservation, "Equator", "6/11/1994", "6/14/94");
   begin(t4);
   if (!commit(t4)) {
      do {t5 = initiate(cancel_flight_reservation, air, "6/11/1994", "6/14/94");
          begin(t5);}
      while (!commit(t5));
      // Compensate for the flight reservation already made
      return 0;}
   // At this point, hotel and flight reservations have both been made

   t5 = initiate(car_reservation, "National", "6/11/1994", "6/14/94");
   begin(t5);
   t6 = initiate(exclusive_car_reservation, "Avis", "6/11/1994", "6/14/94", t5);
   begin(t6);
   if (wait(t5)) {    // Whichever of t5, t6 completes first wins
      abort(t6);
      commit(t5);}
   else commit(t6);

   return 1;  // Activity has completed successfully
}
```