Delegation: Efficiently Rewriting History[†]

Cris Pedregal Martin and *Krithi Ramamritham*
Department of Computer Science
University of Massachusetts
Amherst, Mass. 01003–4610
{*cris,krithi*}*@cs.umass.edu*

Technical Report 1995-090
October 26, 1995

Delegation: Efficiently Rewriting History[†]

Cris Pedregal Martin and *Krithi Ramamritham*
Department of Computer Science
University of Massachusetts
Amherst, Mass. 01003–4610
{*cris,krithi*}*@cs.umass.edu*

Technical Report 1995-090
October 26, 1995

Abstract

The notion of transaction delegation, as introduced in ACTA, allows a transaction to transfer responsibility for the operations that it has performed on an object to another transaction. Delegation can be used to broaden the visibility of the delegatee, and to tailor the recovery properties of a transaction model. Delegation has been shown to be useful in synthesizing Extended Transaction Models.

With an efficient implementation of delegation it becomes practicable to realize various Extended Transaction Models whose requirements are specified at a high level language instead of the current expensive practice of building them from scratch. In this paper we identify the issues in efficiently supporting delegation and hence extended transaction models, and illustrate solutions in two systems: ARIES (which uses UNDO/REDO recovery) and EOS (NO-UNDO/REDO). Since delegation is tantamount to rewriting history, a naïve implementation entails frequent and costly log accesses, and complicates recovery protocols. Our algorithm achieves the effect of rewriting history without rewriting the history, i.e., the log, resulting in implementations that realize the semantics of delegation at minimal additional overhead and incur no overhead when delegation is not used. Besides showing the efficient application to EOS and ARIES, we also show the correctness of the implementation of delegation.

Our work indicates that it is feasible to build efficient and robust, general-purpose machinery for Extended Transaction Models. It also leads toward making recovery a first-class concept within Extended Transaction Models.

Keywords: Extended Transaction Models, Transaction Management, Recovery.

† Supported in part by grants from Sun Microsystems and the National Science Foundation.

Contents

1 Introduction

The transaction model adopted in traditional database systems has proven inadequate for novel applications of growing importance, such as those that involve reactive (endless), open-ended (long-lived), and collaborative (interactive) activities. Various *Extended Transaction Models* (ETMs) have been proposed [10], each custom built for the application it addresses; alas, no one extension is of universal applicability. To address this problem, we have been investigating how to create general-purpose and robust support for the specification and implementation of diverse Extended Transaction Models. Our strategy has been to work from first principles, first identifying the basic elements that give rise to different models and showing how to realize various Extended Transaction Models using these elements, and then proposing mechanisms for implementing these elements.

A first step was ACTA [6], that identified, in a formal framework, the essential components of Extended Transaction Models. In more operational terms, ASSET [5] provided a set of new language primitives that enable the realization of various Extended Transaction Models in an object-oriented database setting. In addition to the standard primitives Initiate (to initialize a transaction), Begin, Abort, and Commit, ASSET provides three new primitives: *form-dependency*, to establish structure-related inter-transaction dependencies, *permit*, to allow for data sharing without forming inter-transaction dependencies, and *delegate*, which allows a transaction to transfer responsibility for an operation to another transaction.

Traditionally, the transaction invoking an operation is also responsible for committing or aborting that operation. Delegation separates these two concerns, so that the invoker of the operation and the transaction that commits (or aborts) the operation may be different. In effect, to delegate is to *rewrite history*, because a delegation makes it appear as if the delegatee transaction had been responsible for the delegated object all along, and the delegator had nothing to do with it.

Delegation is useful in synthesizing Extended Transaction Models because it broadens the visibility of the delegatee, and because it controls the recovery properties of the transaction model. The broadening of visibility is useful in allowing a delegator to selectively make tentative and partial results, as well as hints such as coordination information, accessible to other transactions. The control of the recovery makes it possible to decouple the fate of an update from that of the transaction that made the update; for instance, a transaction may delegate some operations that will remain uncommitted but alive after the delegator transaction aborted. Examples of Extended Transaction Models that can be synthesized using delegate are Joint Transactions, Nested Transactions, Split Transactions, and Open Nested Transactions [6, 11].

Biliris et al. [5] gave a high-level description of how to realize the three new ASSET primitives. Briefly, permit is done by suitably adding the permittee transaction to the object's access descriptor. Form-dependency is done by adding edges to the dependency graph, after checking for certain cycles. Whereas the realization of permit and form-dependency are rather straight-forward, close attention must be paid to logging and recovery issues in the presence of delegation. This is because recovery usually keeps some kind of system history (e.g., log) and delegation is tantamount to *rewriting history* (a delegated object's operations appear to have been done by the delegatee).

Developing a robust, efficient, and correct implementation of delegation is the goal of this paper.

Specifically, to further the goal of providing general purpose machinery to support the specification and implementation of arbitrary Extended Transaction Models, we propose here efficient implementations of delegation based on ARIES [14] and on EOS [4]. Our additions allow the "efficient Rewriting of History." We hence call our implementations ARIES/RH and EOS/RH.

By providing delegation, we add substantial semantic power to a conventional Transaction Management System, allowing it to capture various Extended Transaction Models. We efficiently achieve this expressiveness by carefully "piggy-backing" the delegation-related processing onto the routine processing. During recovery, our algorithm neither adds costly log sweeps to the recovery algorithm, nor does it demand the actual rewriting of history, i.e., the log.

In this paper we argue that:

- Delegation is a powerful, important primitive for realizing Extended Transaction Models. We describe its properties and show how it can be used to manipulate visibility and recovery properties of transactions.
- It is possible to implement delegation in an industrial-strength transaction management system. We illustrate by extending two such systems, ARIES and EOS. We thus obtain the ETM semantics with little loss of efficiency, and when delegation is not used no overheads are incurred. We also demonstrate the correctness of our algorithms.

The remainder of the paper is organized as follows. In Section 2 we describe the properties of delegation and show how it can be used to synthesize some well-known extended transaction models. In section 3 we develop delegation in the context of robust, industrial-grade transaction management systems. First we explain delegation's semantics in terms of rewriting history. We then discuss the needed data structures and describe how we modify both the normal processing and the recovery phases to support delegation, and explain how to apply our algorithm to ARIES and EOS. In section 4 we discuss why our algorithm correctly implements delegation and why it does it efficiently. We review related work in section 5. In section 6 we present our conclusions and discuss future work.

2 Delegation: Concepts, Properties, Examples

In this section we examine the properties of delegation and present its application to extended transaction models. First we introduce some notation, then we explain delegation in terms of visibility and recovery, and then point out some important properties. In the rest of the section we present examples of extended transaction models and show how to synthesize them using delegation.

2.1 What: Concepts and Properties

Here we describe the properties of delegation, introduce notation and state our assumptions.

2.1.1 Assumptions and Notation

- $t, t_0, t_1, t_2, \ldots$ denote ***transactions***; $ob, ob', a, b, \ldots$ denote ***objects*** in the database.
- $delegate(t_1, t_2, update[ob])$ denotes ***delegation*** by t_1 to t_2 of $update[ob]$ where ***update[ob]*** is our one generic operation on object ob. We leave the details of the update unspecified.
- H denotes the **history** of the database, which contains **events** such as *delegate* and *update*, with a partial order indicated $\epsilon \rightarrow \epsilon'$ where ϵ precedes ϵ'. Operation invocations are events.
- ***ResponsibleTr.*** Let transaction t update object ob. We say that t is *responsible for its updates to ob*, when t is in charge of changes to ob. More precisely, $ResponsibleTr(update[ob]) = t$ holds from when t performs $update[ob]$, or is delegated $update[ob]$ until t either terminates or delegates $update[ob]$.
- ***Op_List.*** The dual of *ResponsibleTr* is the *Op_List*. It contains the operations a transaction is responsible for: $Op\_List(t) = \{update[ob] \mid ResponsibleTr(update[ob]) = t\}$

2.1.2 Properties

Pre- and Postconditions. When t_0 executes $delegate(t_0, t_1, update[ob])$, we say that t_0 transfers its responsibility for $update[ob]$ to transaction t_1, i.e.,

- $pre(delegate(t_0, t_1, update[ob])) \Rightarrow (ResponsibleTr(update[ob]) = t_0)$
  t_0 must be the transaction responsible for $update[ob]$ in order to delegate the update.
- $post(delegate(t_0, t_1, update[ob])) \Rightarrow (ResponsibleTr(update[ob]) = t_1)$
  After t_0 delegates $update[ob]$ to t_1, t_1 becomes the responsible transaction for the update.

Operation $delegate(t_0, t_1, update[ob])$ is well formed when t_0 and t_1 are initiated and not terminated, and t_0 is responsible for $update[ob]$.

Commit/Abort of Updates. In the presence of delegation, the fate of updates to an object is not necessarily linked to the transaction which made the updates, but instead it is linked to the fate of the transaction to which the operation was last delegated. For instance, if t_0 does $update[ob]$, then delegates $update[ob]$ to t_1, and t_0 subsequently aborts, the changes t_0 made to ob via $update[ob]$ will still survive if t_1 commits while it is still responsible for $update[ob]$, i.e.,

- $(Commit(t) \in H) \Leftrightarrow (\forall update[ob] \in Op\_List(t), (commit_t(update[ob]) \in H))$
  That transaction t commits means that all of the updates in its Op_List must be committed. Notice that these are the updates for which t is responsible.

- $(Abort(t) \in H) \Leftrightarrow (\forall update[ob] \in Op\_List(t), (abort_t(update[ob]) \in H))$
  That transaction t aborts requires that all of the updates it is responsible for (i.e., those in its Op_List) will be aborted.

The events $Commit(t)$ and $Abort(t)$ denote the commit and abort of transaction t, and $commit_t(update[ob])$ and $abort_t(update[ob])$ indicate the permanence or obliteration of the changes done by $update[ob]$.[1] In the presence of delegation, the changes may have been made by either t or other transaction(s) which eventually delegated $update[ob]$ to t.

Concurrent Delegations. An operation can be delegated only by the transaction that is responsible for it. Since *ResponsibleTr(update[ob])*, is at any given time, unique, only one transaction can delegate an operation at any point in time. Thus, while a history may contain two or more delegations of the same operation by different transactions, the delegations for the same *operation* can not occur concurrently.

Granularity: delegating one operation vs. set of operations. In what we have discussed, a transaction delegates a single operation with a particular invocation of delegate. Delegation of a set of operations in a single invocation can be considered as the atomic invocation of multiple delegations, one for each of the operations in the set. When a transaction delegates an object, it is tantamount to delegating all the operations on that object.

In fact, in our implementation we consider the delegation of objects since in a majority of practical situations that we have come across, delegation occurs at the granularity of objects. Also, in the examples discussed in the next subsection, transactions delegate objects.

Note that it is possible for several transactions to update an object concurrently (say, when the updates commute). Delegation of one such operation by one of the concurrent transactions only delegates that transaction's operation on the object. The other transactions' operations are not affected. Similarly, when a transaction delegates an object, only that transaction's operations on the object are delegated.

Also note that a transaction can perform operations on an object even after it has delegated (an operation on) that object. Of course, since after delegation the system considers the delegated operations to have been done by the delegatee, a transaction's operation may conflict with one of its own — one which has been delegated.

[1]How the semantics of commit and abort are realized depends on the particular recovery method, for example, NO-UNDO/REDO.

2.2 Why: Synthesizing Extended Transaction Models – Examples

In this section we motivate delegation through examples of its application in the synthesis of two extended transaction models, split/join transactions and nested transactions.

Inheritance in nested transactions is an instance of delegation. Delegation from a child transaction t_c to its parent t_p occurs when t_c commits. This is achieved through the delegation of all the changes done by t_c to t_p when t_c commits. That is, all the changes that a child transaction is responsible for are delegated to its parent when it commits.

A transaction can delegate at any point during its execution, not just when it aborts or commits. For instance, in Split Transactions [16], a transaction may *split* into two transactions, a splitting and a split transaction, at any point during its execution. A splitting transaction t_1 may delegate to the split transaction t_2 some of its operations at the time of the split. Thus, a split transaction can affect objects in the database by committing and aborting the delegated operations even without invoking any operation on the objects.

In the remainder of this section we show the code for split and nested transactions, synthesized using delegation and the other ASSET primitives. Other transaction models using delegation include Reporting Transactions and Co-Transactions described in [7, 8]. The former periodically reports to other transactions by delegating its current results. In the latter, control is passed from one transaction to the other transaction at the time of delegation.

2.2.1 Split Transactions

In the split transaction model [16] a transaction t_1 can *split* into two transactions, t_1 and t_2. Operations invoked by t_1 on objects in a set *ob_set* are delegated to t_2. t_1 and t_2 can now commit or abort independently. Conversely, two transactions, say t_1 and t_2 can *join* to form one transaction t_1.

Consider the following code used by t_1 to split off transaction t_2 (the code for t_2 is that of function f.)

```
t2 = initiate(f);
delegate(self(), t2, ob_set);           // self returns t1
begin (t2);
```

t_2 can join t_1 by executing:

```
wait (t2);
delegate (t2,t1);                       // t2 delegates *all* objects
```

2.2.2 Nested Transactions

Nested transactions are among the first extended transaction models; they are discussed by Moss [15]. A nested transaction 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, and 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 creating a conflict. Abort semantics for both root transactions and subtransactions are similar to abort semantics in atomic transactions. Commit, however, has different semantics for the root and the subtransactions. When a subtransaction commits, the objects modified by it are made accessible to its parent transaction. The effects on objects are only made permanent on the commit of the topmost root transaction.

We illustrate how nested transactions are translated into the ASSET primitives with a simple two-level example of trip arrangements.

```
tid t;
t = trans {
    trans { airline_res(); }
    trans { hotel_res();   } }
```

If the airline reservation fails, then the trip is canceled. If the hotel reservation fails, the trip is canceled too, and the effects of the airline reservation should not be made permanent. The nested transaction above is translated into:

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

where the function `trip` is

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

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

We assume that `t1` and `t2` will each abort if they are unsuccessful. If they succeed, they delegate their updates to `t`. Otherwise any updates made so far are discarded. Note that after it has delegated all its changes, the fate of a reservation transaction does not matter.

3 How: Rewriting History Efficiently

In this section we discuss how to efficiently implement delegation and present our algorithm RH (rewrite history), as follows. In 3.1 we introduce the operational semantics of delegation in the context of a generic Database Management System (DBMS). In 3.2 we examine alternative solutions and give an overview of our algorithm. The remainder of the section explains the algorithm in detail: we present the data structures involved in 3.3, then we describe in 3.4 what RH does during normal processing, and finally we tackle the recovery. We begin 3.5 with short overviews of EOS's recovery (whose NO-UNDO/REDO protocol requires one forward pass on the log during recovery) and ARIES's (whose UNDO/REDO protocol requires two passes, one forward and one backward, over the log). Then we discuss how RH's recovery realizes delegation efficiently using the same passes over the log as EOS (one pass) and ARIES (two passes) respectively.

3.1 Operational Semantics

In a DBMS the log *is* the system's history, as it contains the records of all updates, transactional operations, etc. The idea of delegation is to *rewrite history*, selectively *altering the log.* Suppose that $delegate(t_1, t_2, ob)$ is the first delegation of ob by t_1. Applying this delegation can be visualized as iterating through the log into the past, modifying the records pertaining to ob, so that each record of an access to ob by t_1 will now show that the access was done by t_2.

The log is a list held in stable storage, whose elements are identified by monotonically increasing values of the *Log Sequence Number* (LSN). During normal execution, the only valid operation is appending a log record to the end of the log (with the corresponding increment of the current Log Sequence Number). During recovery, the log can be rolled back and replayed, by going to the LSN of the last checkpoint and extracting, sequentially, the records from there on.

Figure 1 is the operational description of delegation in terms of the log. There, currLSN indicates the LSN being operated on in the current iteration. Records have a PrevLSN field, that contains the LSN of the previous record for the same transaction. The chain formed by PrevLSN pointers of log records of a transaction is called *Backward Chain* (see 3.5.2). The **`delegate`** record is a new type that records a delegation, with pointers to the previous records of both the delegator and delegatee (see section 3.3).

In figure 1 we use the following operations on the log:

```
currLSN ← LSN of delegate record
while currLSN is not the initiate record for $t_1$
    if currLSN is an update to ob by $t_1$
        then setTransID(currLSN,$t_2$);
    currLSN ← prevLSN(currLSN,$t_1$)
```

Figure 1: Operational semantics of $delegate(t_1, t_2, ob)$.

prevLSN(LSN, t_1) which returns the Log Sequence Number of the previous (most recent) log record written by t_1 (i.e., before, or to the left of LSN).

setTransID(LSN,t_2), which does $log[LSN].TransID \leftarrow t_2$, making the record appear as if it had been written by the transaction t_2.

The fields in a log record are: LSN (log-sequence number), Type (update, delegation, commit, etc.), Trans-ID (the ID of the transaction that created the record), and Data. For delegate records there also exist two LSN pointers to the delegator and delegatee (see 3.3).

Example. Consider the log fragment (see figure 2):

$\ldots update(t_1, a), update(t_2, x), update(t_1, b), update(t_1, a), update(t_2, y)$

After the application of delegate(t_1, t_2, a), the log looks like:

$\ldots update(t_2, a), update(t_2, x), update(t_1, b), update(t_2, a), update(t_2, y)$

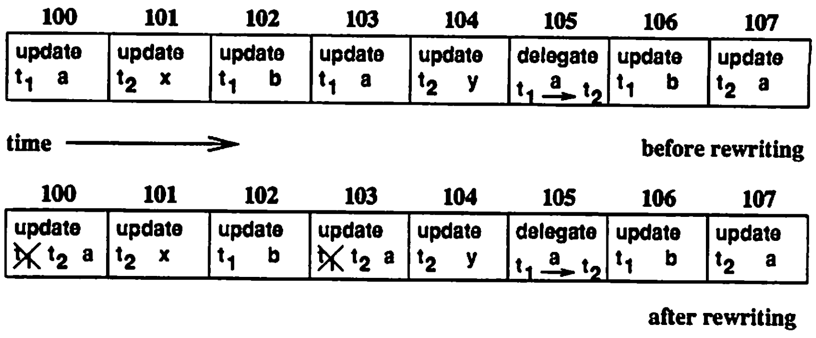

Figure 2: Delegation Log Example

3.2 Implementing Delegation Efficiently

The idea of rewriting history by modifying the log is simple, but its implementation is not. The naïve implementation of the algorithm in figure 1 would be to apply each delegation to the log as it is issued. That is, every time a delegation is issued, the system traverses the log backwards modifying the records pertaining to the object being delegated. This "eager" approach carries high performance costs, and is also hard to prove correct. The performance penalty is due to the random nature of the accesses (as opposed to the usual append-only to logs), and the fact that a single delegation will generate many accesses, in principle sweeping the whole log [17]. Ensuring recovery correctness is hard because we manipulate the log outside the usual append-only mode, complicating the model with extra data,[2] whose integrity in the face of crashes is not guaranteed by the standard recovery algorithm and must be ensured ad-hoc.

A better approach may be to use a "lazy" algorithm that defers the alteration of the log to recovery. This is based on the observation that during normal processing it is not necessary to have the delegations applied to the log. The algorithm can keep track of the effect of delegations in volatile data structures, and log the delegations to have the necessary information after a crash. It modifies the log – rewrites history – during recovery, which manipulates the log anyway, based on the information on the log about updates and delegations. For example, in UNDO/REDO (see section 3.5.2), the algorithm uses the logged information on updates and delegations to reconstruct the information about delegations the during analysis/redo (forward) pass. Then in the undo (backward) pass, it modifies the log as suggested in section 3.1 and figure 3, moving records from the delegator's backward chain to the delegatee's, and rewriting the record to make it appear as if created by the delegatee.

Although this is workable, it still suffers from drawbacks. Because it modifies the log in other than append mode, issues of correctness in the face of failure and performance must be addressed. It is possible to solve the correctness problem by ensuring that each BC switch is done atomically.[3] The performance, however, is inherently hostage to the way the log is accessed. Recall that in general the log does not fit into volatile storage. The buffering can result in thrashing, as the algorithm needs to jump over possibly large extensions to follow backward chain pointers.[4]

To avoid these pitfalls, we propose RH, a "lazy" algorithm for rewriting history that does not modify the log. We give a brief overview here. As pointed out before, delegation can be supported easily during normal processing. During normal processing, we use a volatile table to keep track of which objects are updated by which transactions. When a delegation happens, we change the corresponding object binding, and log delegations to be able to reproduce the change after the crash. During recovery, on encountering delegations during the log sweeps, we reconstruct the bindings between operations on objects and transactions, but do not actually rewrite the log records. We "rewrite the history" of the system not by modifying the log, but by *interpreting* the log during recovery according to the delegations. That is, we obtain the desired

[2]Extra data: information accessed by transactions that is not part of the database schema; for example, the log, the system clock, wait-for graph. Gehani et al. [11] discuss the issues of correctness with extra data.

[3]It is easier to tolerate unusual log manipulations during recovery than during normal processing.

[4]The problems of this approach are discussed in detail in [17].

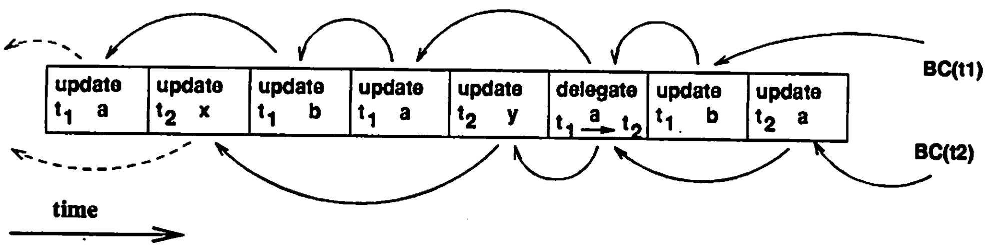

Figure 3: Backward Chains in the Log

semantics – rewrite of the history according to the delegations – without having to actually rewrite the log.

In the remainder of this section we explain RH in detail, in the context of two conventional recovery protocols: NO-UNDO/REDO and UNDO/REDO. We present the data structures, and we indicate how the normal processing keeps the tables up to date. We then examine recovery processing, first with the NO-UNDO/REDO protocol (as done in EOS/RH), and finally with the more complicated UNDO/REDO protocol (as done in ARIES/RH).

3.3 Data Structures

For each transaction, we must know which operations on which objects it is responsible for, i.e., its Op_List. We use and augment *Transaction List* and each transaction's *Object List* found in conventional DBMSs. We also add a `delegate` type log record.

Tr_List. We use the standard *Transaction List* Tr_List [2, 12, 14] that contains, for each Trans-ID, the LSN for the ***most recent*** record written on behalf of that transaction, and, during recovery, whether a transaction is a *winner* or a *loser* (see 3.5.1 and 3.5.2). Notice that for each transaction t, Tr_List(t) contains the head of the backward chain BC(t). Figure 3 shows the backward chains in the delegation example of section 3.1.

Ob_List. Associated with ***each*** transaction t there is an *Object List* Ob_List(t). E.g. (see figure 4), Ob_List(t_1) lists the objects t_1 is currently accessing.[5] More precisely, in terms of Op_List (see 2.1): Ob_List(t) $= \{ob \mid \exists update(t_0, ob) \in Op\_List(t)\}$. Notice that a certain object may appear in more than one Ob_List, but the associated updates will be non-intersecting.

We augment the conventional Object List with a *deleg* field for each object that indicates whether this object was acquired through a delegation and from whom. Specifically, Ob_List(t_2)[a].*deleg* $= t_1$ if a was delegated by t_1 to t_2, or Ob_List(t_2)[x].*deleg* $= t_2$ if x was created or accessed originally by t_2. We also associate with each object the *scope* of its delegation.

Scope. Delegations have an associated scope. To see why, consider a transaction t that

[5]For example, in some implementations Ob_List may also have pointers to locks on the objects.

object	deleg by trans ID	scope
~~a~~	~~t_1~~	
b	t_1	

Ob_List (t1)

object	deleg by trans ID	scope
a	t_1	*100*
x	t_2	
y	t_2	

Ob_List (t2)

Figure 4: Object Lists after applying example delegation.

updates *ob*, then t delegates *ob*[6] to t_1, then t again updates *ob* and t delegates it to t_2. If t_1 commits and t_2 aborts, the first update (delegated to t_1) must persist, whereas the second (to t_2) must be undone. It is clear then that the second delegation's scope begins after the first delegation. The first delegation's scope goes back to t's first update on *ob* (assuming t was not itself delegated *ob*). In other words, the *scope* of delegation is the extent on the log to which the delegation applies, and it is indicated by a Log Sequence Number.

Delegate Log Records. We also introduce a new log record type: **delegate**. Its type-specific fields record the two transactions and object involved in the delegation. Besides the conventional records, in this type the field Data contains the delegated object ID and delegatee Transaction ID (see 3.1). To facilitate the undo pass, for ARIES (see 3.5.2) the record also includes DelegatorPrevLSN (link to the delegator's Backward Chain) and DelegateePrevLSN (to the delegatee's Backward Chain). The other record types are not modified, and are as discussed in 3.1.

3.4 Normal Processing

We focus on the changes entailed by delegation. Our delegation algorithm augments the normal processing. We describe how different events are processed, as follows:

delegate(t_1, t_2, ob)

- ▷ WELL-FORMED? Check that the delegation satisfies preconditions stated in 2.1.2.
- ▷ LOG DELEGATION. Write a **delegate** log record containing the delegating transaction's id t_1, the delegatee's t_2, and the object name *ob*.

  In ARIES/RH, we also link this log record into t_1's and t_2's backward chains, i.e.,
  $LOG[currLSN].DelegatorPrevLSN \leftarrow \mathrm{BC}(t_1).PrevLSN$ and
  $LOG[currLSN].DelegateePrevLSN \leftarrow \mathrm{BC}(t_2).PrevLSN$.
- ▷ UPDATE OBJECT'S SCOPE. The delegatee inherits the scope for this object; the delegator starts a new scope for future operations on this object:

[6]Remember that *delegate*(t, t_1, ob) really delegates operations, i.e., t's updates to *ob*.

Ob_List$(t_2)[ob].scope \leftarrow$ Ob_List$(t_1)[ob].scope$, followed by
Ob_List$(t_1)[ob].scope \leftarrow$ CurrLSN.

Remark: notice that the scope indicates how far back the delegated updates (received by the delegatee transaction t_2) may go in the log. In particular, the scope may reach updates done before the beginning of t_2. The scope is essential for the backward pass of the recovery algorithm (3.5).

▷ UPDATE OBJECT LISTS. Transfer *ob* from Ob_List(t_1) to Ob_List(t_2). Record that *ob* was delegated by t_1, i.e.: Ob_List$(t_2)[ob].deleg \leftarrow t_1$.

update(t, ob)

▷ INITIALIZE SCOPE? If $ob \notin$ Ob_List(t) then create entry in Ob_List and set initial scope Ob_List$(t)[ob].scope \leftarrow$ CurrLSN.

commit(t)

▷ COMMIT. Commit operations in Op_List(t).
▷ LOG COMMIT RECORD.

abort(t)

▷ ABORT. Undo updates on objects in Ob_List(t); log undos.[7]

Notice that any object that had been delegated *by* the aborting transaction will no longer be in the transaction's Ob_List.

▷ LOG ABORT.

The other transactional events are processed as usual [12, 14].

3.5 Recovery

After a crash, the transaction system must do some recovery processing to return to a consistent state. This entails restoring the state from a checkpoint (retrieved from stable storage), and using the log (also from stable storage) to reproduce the events after the checkpoint was taken. Several strategies exist to do this, depending on how the Undo Rule and the Redo Rule are enforced [2]. We consider two approaches: that of EOS [4], which uses a NO-UNDO/REDO strategy, and that of ARIES [14], which does UNDO/REDO. For simplicity of presentation, we ignore checkpoints from now on, but it is easy to see how data structures can be rebuilt using checkpoints instead of going back to the beginning.

Crash is the event that represents a failure; *RecoveryComplete* is the event that appears in the history to indicate that the recovery is complete.

Winners is the set of transactions that had committed before the crash. The recovery algorithm ensures that their updates will be reinstalled. *Losers* is all the other transactions

[7]In EOS, simply discard updates.

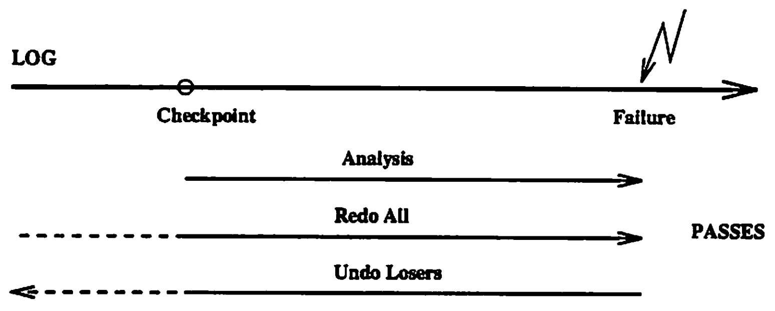

Figure 5: ARIES passes over the log

that were active but had not committed, or had aborted, before the crash. Their updates must be undone.

The rest of the section is organized as follows. We first give an overview of EOS and ARIES, to establish context and terminology, so that we can explain RH by describing how it modifies EOS or ARIES. After that we present the forward pass of RH, which is common to both EOS/RH and ARIES/RH. We establish the state after that pass, which completes EOS/RH. Finally, we describe the backward pass, used in ARIES/RH.

3.5.1 Overview of EOS

EOS uses a NO-UNDO/REDO protocol. To avoid having to undo changes to the database, it avoids applying those changes until the transaction that made them is ready to commit. That is, only updates from committed transactions are ever applied; updates from aborted transactions are never logged. As a result, when recovering from a crash, all the log records correspond to committed transactions, so EOS applies (redoes) them in a single *forward* pass. Transactions that had not committed before their crash have disappeared without a trace, as their changes were never logged. Notice that for EOS, all logged transactions are *winners*.

3.5.2 Overview of ARIES

ARIES uses an UNDO/REDO protocol, which means that after a crash, some updates will be undone and some redone, according to whether the responsible transaction is a *winner* or *loser*. ARIES scans the log in one or two *forward* passes: analysis and redo; and then a *backward* pass: undo. See figure 5.

The *analysis* pass starts at the last checkpoint, updates the information on active transactions and dirty pages up to the end of the log, and also determines the "loser" transactions, to be rolled back in the undo pass. The *redo* pass *repeats history*, writing to the database those updates that had been posted to the log but not applied before the crash. This re-establishes the state of the database at failure time, including uncommitted updates. Because some ARIES variants merge the analysis and redo passes in a single forward pass, ARIES/RH relies on a single forward pass to add delegation.

The *undo* pass rolls back all the updates by loser transactions in reverse chronological order starting with the last record of the log.

To facilitate its UNDO/REDO recovery, ARIES keeps, for each transaction, a *Backward Chain* (BC) linking the transaction's records in the log. That is, all the log records pertaining to one transaction form a linked list, beginning with the most recent one, which is accessible through the Tr_List. By following the BC, ARIES's recovery avoids repeating undos, as it can insert *compensation log records* (CLRs) to indicate how to undo an action or whether to skip an already undone action.

In terms of Backward Chains, applying $delegate(t_1, t_2, ob)$ is tantamount to removing the subchain of records of operations on *ob* from BC(t_1), merging it with BC(t_2).

3.5.3 Forward Pass

Both EOS and ARIES start with a forward pass that finds out which transactions were active, and which committed, before the crash. EOS and ARIES also use the forward pass to *redo* logged updates.[8] At the end of this pass the *Object Lists* are up to date.

We describe RH by its treatment of the log records that matter to delegation. Other records are processed as usual. Before the first pass of recovery starts, $Winners = Losers = \phi$. For brevity we omit some details already explained for the normal processing.

$begin(t)$
- ▷ LOSER BY DEFAULT. Consider t a loser by default, i.e. $Losers \leftarrow Losers \bigcup \{t\}$.

$update(t, ob)$
- ▷ VERIFY. Check that $ob \in$ Ob_List(t).
- ▷ REDO. Redo $update(t, ob)$.

$delegate(t_1, t_2, ob)$
- ▷ LOOKUP. Verify that $ob \in$ Ob_List(t_1).
- ▷ TRANSFER. Move *ob* from Ob_List(t_1) to Ob_List(t_2) updating its scope.

$commit(t)$
- ▷ COMMIT. Commit t.
- ▷ WINNER. Declare t as a winner: $Winners \leftarrow Winners \bigcup \{t\}$ and $Losers \leftarrow Losers - \{t\}$.

EOS In the case of EOS (NO-UNDO/REDO), recovery is now complete. Notice that for EOS there are neither abort records nor updates from aborted transactions that need to be undone; the *Losers* set is not necessary. For EOS, delegations have been made effective during normal

[8]ARIES may use two forward passes; see 3.5.2.

processing, as the committed objects (i.e., posted updates) are exactly those that were in *Object List* of transactions that committed before the crash.

ARIES In the case of ARIES (UNDO/REDO), each active transaction which had not committed before the crash is now in *Losers*. Note that this includes transactions that aborted before the crash.

After the Forward Pass we have updated the following.

- Ob_Lists are restored to their state before the crash, for all transactions.
- *Winners* has all the transactions whose updates must survive (i.e., which had committed before the crash). *Losers* has those whose updates must be obliterated.
- *LoserObjs* includes all objects in the Ob_Lists of loser transactions. It is computed after the pass ends (for ARIES/RH only). That is, $LoserObjs = \bigcup_{t \in Losers} Ob\_List(t)$

3.5.4 Backward Pass

ARIES undoes exactly all the updates *done* by the loser transactions. Given the backward chains that link records of updates by each transaction, it is a matter of following the BCs for each of the loser transactions, undoing *all* their updates. ARIES does this by continually taking the maximum LSN for an outstanding undo, ensuring monotonically decreasing (by LSN) accesses to the log, with the attendant efficiencies.

With delegation, however, this approach is not possible. What we need to achieve is the undo not of all the updates by a loser transaction, but instead of all the updates that were ultimately delegated to a loser transaction. Thus, we would need a backward chain linking those updates, but constructing and maintaining such a structure is complicated and expensive. The next alternative is to scan *all* log records backwards, identifying the updates that need to be undone; this is also unacceptably inexpensive.

Fortunately, the necessary information to achieve the result efficiently is already present in the object lists of the loser transactions, in the form of scope and back pointers. The updates that need to be undone form a *closure*, that we can cover efficiently in a manner similar to that of ARIES, i.e., with monotonically decreasing LSN values. In the rest of the section we discuss how both the undo and delegation are integrated in the backward pass of RH. The only records that require special processing are update and delegation; all others are processed as in ARIES.

During the backward pass we apply the delegations by undoing the updates to *loser objects*. Compare with conventional ARIES that undoes all the updates made by *loser transactions.*[9] For each undone update we write as in ARIES its *Compensation Log Record* (CLR). This record

[9]In ARIES, all loser objects are the objects updated by loser transactions, so our solution reduces to ARIES when there is no delegation.

```
$ToCheck \leftarrow \{Tr\_List(t) \mid t \in Losers\}$
LSN $\leftarrow$ max($ToCheck$); $ToCheck \leftarrow ToCheck -$ {LSN}
while $ToCheck \neq \phi$ and LSN $\geq minscope$
    if LOG[LSN] = $update(t, ob)$ and $ob \in LoserObjs$ and Ob_List($t$)[$ob$].$scope \leq LSN$
        then undo($update(t,ob)$)
            CLR.PrevLSN $\leftarrow$ LOG[LSN].PrevLSN and other undo information ...
            writeLog(CLR)
    else if LOG[LSN] = $delegate(t_1, t_2, ob)$
        then $ToCheck \leftarrow ToCheck \cup$
            {LOG[LSN].delegatorPrevLSN, LOG[LSN].delegateePrevLSN}
    LSN $\leftarrow$ max($ToCheck$); $ToCheck \leftarrow ToCheck -$ {LSN}
```

Figure 6: Backward pass of ARIES/RH.

allows ARIES (and ARIES/RH) to avoid undoing an update repeatedly should a crash occur during recovery.

Tr_List contains, for each transaction, the LSN of the most recent record written by the transaction. Tr_List is maintained during normal processing and reconstructed during the forward pass. As we process a record for e.g. t_1, we update Tr_List(t_1) to point to the predecessor of the update record processed. Through this we are applying the undo in reverse chronological order by following the backward chains (see 3.3 and 3.5.2).

Before presenting the algorithm for the backward pass (figure 6), we need some definitions. The *Minimum Loser Scope* is an LSN which we note *minscope*, defined:

$minscope = min\{$ Ob_List$(t)[ob].scope \mid ob \in LoserObjs\}$.

Minscope is the smallest value of LSN for which there are updates that have been delegated to some loser transaction, and therefore must be undone. It can be computed before starting the backward pass; see the remark in section 3.4.

We also keep *ToCheck*, a set of LSNs that point to records that *may* need to be undone. *ToCheck* is initialized to the last records of the *Losers* transactions, and the algorithm progresses by removing the maximum LSN, possibly undoing it, if it is an update. When the record is a delegate, we add the LSNs of the previous records for both transactions, to ensure that all possible loser objects are reached. We need to include the preceding record of the delegator transaction to be able to reach the records of the updates for which the delegatee just became responsible through the delegation, which appear on the delegator's backward chain. We need to include the preceding record of the delegatee as there may be other updates the delegatee is responsible for in its own backward chain.

Although the set *ToCheck* may grow or shrink the loop always terminates. If there are no more log records to examine (reachable through the backward chains), *ToCheck* will be empty. If there remain records to examine, but they are outside the scope of delegation, LSN will become smaller than *minscope*. Notice that, regardless of the size of *ToCheck*, LSN *always* decreases, as it is extracted as the maximum of a set of log sequence numbers that can only decrease because we are following backward chains.

4 Discussion

In this section we discuss two issues with regard to the implementation. First we analyze aspects of the algorithm presented in section 3 to argue that it implements delegation correctly. In the second part, we look at the implementation from the point of view of efficiency.

4.1 Correctness

Correctness in a recovery protocol means that after the system completes recovery, its state is consistent with the transaction correctness rules. Specifically, ARIES guarantees that all operations by *loser* transactions will have been rolled back completely (their effects obliterated) and all by *winner* transactions will be committed (their effects guaranteed to be permanent). We show in this section that our modifications comply with this requirement when it is rephrased to include delegation. That is, operations delegated to loser transactions will be aborted, and operations delegated to winner transactions will be committed. Naturally, operations originally carried out by a transaction and not subsequently delegated, are treated as in the conventional case.

After the event *Crash*, we initiate recovery, which ends with the event *RecoveryComplete*. Between *Crash* and *RecoveryComplete* all events are generated by the recovery system. For simplicity, we ignore checkpoints and assume that the system restarts from the beginning.

A brief recapitulation is in order. The forward pass reads but does not write anything to the log. It redoes the updates present in the log, and constructs the sets *Winners* of transactions whose updates will survive after the recovery (for EOS, all that are logged). This is enough for EOS as it does not need to undo anything, and the delegations are reflected by the fact that winners commit their *Object List* objects during normal processing. For ARIES/RH it also records the *Losers*, i.e., active transactions that did not commit before the crash, and computes *LoserObjs* after the forward pass. The backward pass reads the log, interpreting it according to the delegations, and undoes updates on loser objects.

In the remainder of this section, we characterize loser and winner transactions and their associated updates, and show the correctness of the algorithm, to wit, that all loser updates get undone and all winner updates get redone.[10]

Winners, Losers, LoserObjs.

- $t \in Winners \iff (Commit(t) \rightarrow Crash)$
  t is a winner if it committed before the crash.
- $t \in Losers \iff (Begin(t) \rightarrow Crash \wedge \not\exists Commit(t) \in H)$
  t is a loser if it was active but did not commit before the crash.

  Losers: an active transaction is by default a loser.[11] If there is a commit record before the crash, its transaction is moved to *Winners*. Note that these sets are disjoint.
- $LoserObjs = \bigcup_{t \in Losers} \text{Ob\_List}(t)$ i.e., $ob \in LoserObjs \Rightarrow \exists t \in Losers : ob \in \text{Ob\_List}(t)$
  LoserObs is the set of all objects for which there is a loser transaction that is responsible for an update to that object. This means that a loser object has at least one update that will be undone.

Delegation Closure. If t_n is responsible for an update, t_n must have done the update itself $(n = 0)$ or received it from t_0 through a sequence of delegations:

$update(t_0, ob) \in Op\_List(t_n) \implies (\exists(n \geq 0), t_0, t_1, , ..., t_{n-1}, t_n$ **such that**

that is, if t_n is responsible for the update, then there is a sequence of transactions, starting with t_0, the transaction that did the update, and ending with t_n, such that

$[update(t_0, ob) \rightarrow delegate(t_0, t_1, ob) \rightarrow ... \rightarrow delegate(t_{n-1}, t_n, ob)] \wedge$

$[\not\exists i \ (0 < i < n), t_x$ **such that** $delegate(t_{i-1}, t_i, ob) \rightarrow delegate(t_i, t_x, ob) \rightarrow delegate(t_i, t_{i+1}, ob)]$

t_0 delegated the update to t_1, and so on, until finally t_n received it in the last delegation; and for each t_i, t_{i+1} is the first transaction to which t_i delegates ob, i.e., there is no other intervening delegate (to, say, transaction t_x) of that update.

That is, if t_n is responsible for an update, there is a sequence of delegations that links the original update log record to t_n.

This holds by induction on n. The base case, for $n = 1$, is immediate from the algorithm that applies delegation during normal processing and the forward pass of recovery.[12] Specifically, each time a delegation is encountered, the object is passed along with its scope. The scope defines uniquely the updates being delegated (see remark in section 3.4). and the normal processing and forward pass of the recovery. For the inductive case, note that the scope is never modified for the delegatee, so it remains the same through the delegations.

Correctness: Redo and Undo

1. $(\forall t \in Losers \ \forall update(t_0, ob) \in Op\_List(t))(Undo(update(t_0, ob)) \rightarrow RecoveryComplete)$
   All updates ultimately delegated to a loser transaction are undone.

[10] *Winners* and *Losers* is of interest for UNDO/REDO (ARIES).
[11] For EOS this is irrelevant since EOS does not undo.
[12] For $n = 0$ it reduces to no delegation and holds trivially.

2. $(\forall t \in Winners\ \forall update(t_0, ob) \in Op\_List(t))(Redo(update(t_0, ob)) \rightarrow RecoveryComplete)$
   All updates ultimately delegated to a winner transaction are redone.

We first discuss (1). If $update(t_0, ob)$ ends up in the *Op_List(t)* and $t \in Loser$, it means that the update is reachable from the last record of t (which is in $Tr\_List(t)$) through a backward chain consisting of segments BCs of all the transactions in the chain, joined by the corresponding delegations. Then, by the algorithm in figure 6, we have that $ob \in ToCheck$ initially, and the record for the update will be added to *ToCheck* and eventually checked.

We prove the redo (2) by contradiction. First, notice that the update is redone in the forward pass. Now let us see that it does not get undone. We proceed by contradiction. The delegation closure applies here too (as it does not depend on the fate of the final transaction), so there must be a chain of BC segments and transactions back to the original log record. If the update were undone, it means that there is a path to its record from a loser transaction (as the initial set *ToCheck* consists exclusively of the last records of loser transactions) back to it through delegation. But that means that the responsible transaction of t was a loser, contradicting that it was a winner update. *Q.E.D.*

4.2 Efficiency

We claim RH is efficient in the following senses:

- **No delegation, no overhead.** When either EOS/RH or ARIES/RH does not use delegation, RH reduces to the original algorithms, so no penalty is incurred due to the provision of the extra functionality.
- **Normal processing: low, linear overhead.** Posting one delegation during normal processing has the cost of adding a log entry and updating the object bindings. The cost of delegations is linear in the number of operations delegated. For instance, the updating of Ob_Lists for a delegation is linear in the length of the Ob_Lists.
- **Recovery: low overhead.** The costs of the recovery passes are similar to those of conventional recovery systems. For all operations, supporting delegation only entails costs at most linear in the number of delegated operations (see previous item). Also, recovery costs are dominated by disk log accesses, which RH does as efficiently as EOS or ARIES. For instance, on the backward pass, log records are visited at most once and in strict decreasing order, as in ARIES. allowing for the usual optimizations.

The first two points follow from the fact that RH only adds some fields to data structures that are already updated by the conventional algorithm. When there is no delegation, these fields are just left undefined. When there is delegation, it adds the constant time of logging the delegation operation and updating, for the delegator and delegatee transactions, their Ob_Lists by moving as many objects as are delegated (hence the linearity). This entails lookup/updates to the transactions' Ob_List, which resides in main memory and can be organized efficiently to have very low lookup/update cost. At transaction termination the Ob_List can be simply discarded.

As for recovery, RH's forward pass incurs the same overhead as EOS or ARIES do to reconstruct transactional data structures and redo updates. Again, the only additional information that is collected is piggy-backed in those data structures. No special sweep of the log is required: RH obtains its information during the same accesses as the conventional algorithms. Specifically, the forward pass of recovery is only different from ARIES in its processing of update (there is an extra check for $ob \in$ Ob_List(t)) and delegate (same check and the move from one Ob_List to the other). Thus, RH adds neither extra log sweeps, nor costs proportional to the length of the log, as it uses the same sweep(s) of the log as EOS or ARIES to reconstruct the delegation information.

We expect the Ob_List to be much smaller than the log being analyzed, and it resides in main memory. Thus the cost of accessing Ob_List is small compared to bringing the log from stable storage, the dominant cost during recovery.

The backward pass of recovery reads the log in much the same way as ARIES, by continually taking the maximum LSN that needs to be undone (in ARIES) or *may have* to be undone (in ARIES/RH). To compare with ARIES, we need only examine the costs for processing update and delegation records (the rest are just as in ARIES). For each update, we do a lookup in *LoserObjs*, a check of delegation scope, to decide whether to undo it, and possibly write a Compensation Log Record. For each delegate, we just add two LSNs to the *ToCheck* set. Both have costs comparable to the usual treatment of undoing updates.

In summary, RH (delegation) algorithms add minimal overheads to support delegation.

5 Related Work

We have benefited from insights gained in an effort with goals closer to ours: the work at GTE Labs on Transaction Specification and Management Environment (TSME, [9]). The architecture of TSME consists of a Transaction Specification Facility that understands TSME's transaction specification language, and drives the Transaction Management Mechanism which configures the run-time system to support a specific Extended Transaction Model. The Transaction Management Mechanism is programmable, but uses templates to describe existing extended transaction models, and also to drive the incorporation of only the components necessary for a given Extended Transaction Model. It is a toolkit approach, in which certain expressions in the specification language are mapped to certain configurations of *pre-built* components, so it approaches the problem at a coarser grain. This may allow for initial gains in performance, but we believe that the use of language primitives is a richer and more flexible approach.

The recent work of Barga and Pu [1], also inspired in part by ACTA, explores another modular approach, based on the ideas of metaobject protocols [13], and incorporates some elements of the TSME approach and some of our language-based approach.

Also related is the work on the ConTract model [20]. In ConTract, a set of steps define individual transactions; a script is provided to control the execution of these transactions. But ConTract scripts introduce their own control flow syntax, while ASSET introduces a small set of transaction management primitives that can be embedded in a host language.

Other related work also includes Structured Transaction Definition Language [3], a persistent programming language geared to portability and the integration of legacy applications. Its emphasis, however, is in Application Programming Interfaces conforming to existing conventional transactional technology.

Finally, the idea of rewriting history is a natural extension of the repeating history paradigm of ARIES [14] and is a generalization of ARIES/NT [19], an extension to ARIES for nested transactions [15].

6 Conclusions

Recent work has produced many Extended Transaction Models (ETMs), but each has its own tailor-made implementation. With delegation (and the other two ASSET primitives, permit and form-dependency [5]) we believe we can offer the flexibility to synthesize a wide range of ETMs at a performance comparable to that of tailor-made implementations. Delegation, by allowing changes in the visibility and recovery properties of transactions, is a very useful primitive for synthesizing Extended Transaction Models. Our work builds on the formal foundation provided by ACTA [6, 7, 8], and the primitives introduced in ASSET [5].

The main contribution of this paper is the concept of *rewriting history* (RH), designed to achieve the semantics of delegation in an efficient and robust manner. We believe that this work forms a crucial step towards the flexible synthesis of ETMs:

- By casting delegation in terms of rewriting history, we were able to express the issues of delegation in terms amenable to the specification of a recovery algorithm.
- We showed how to achieve RH in the context of two practical systems (EOS and ARIES), suggesting the practical implementability of delegation. As indicated in section 4, the cost of delegation in RH is very low, and its support incurs no cost at all when delegation is not being used.
- We have also demonstrated the correctness of our implementation, showing that it satisfies the desired transaction properties in the presence of delegation.

We are currently implementing RH within EOS. We will continue investigating the broader issues of providing robust, efficient, and flexible transaction processing. In particular, we are interested in making recovery a first-class concept within transaction management and in providing a variety of recovery primitives to a transaction programmer so that different recovery requirements and recovery semantics can be achieved flexibly.

References

[1] Roger S. Barga and Calton Pu. A Practical and Modular Implementation of Extended Transaction Models. In *Proceedings of the 21th International Conference on Very Large Data Bases*, September 1995.

[2] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. Concurrency Control and Recovery in Database Systems. Addison Wesley, Reading, Mass. 1987.

[3] Philip A. Bernstein, Per O. Gyllstrom, and Tom Wimberg. STDL – A Portable Language for Transaction Processing. In *Proceedings of the 19th International Conference on Very Large Databases*, pages 218–229, Dublin, 1993.

[4] A. Biliris, E. Panagos. EOS User's Guide AT&T Bell Labs Report, May 1993.

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

[6] P. K. Chrysantis, and Krithi Ramamritham. Synthesis of Extended Transaction Models using ACTA. *ACM Trans. on Database Systems*, September 1994.

[7] P. K. Chrysanthis. *ACTA, A Framework for Modeling and Reasoning about Extended Transactions.* Computer Science TR 91-90. PhD thesis, Department of Computer and Information Science, University of Massachusetts, Amherst, Mass., September 1991.

[8] P. K. Chrysantis, and Krithi Ramamritham. Delegation in ACTA as a Means to Control Sharing in Extended Transactions. IEEE Data Engineering, 16(2): 16-19, June 1993.

[9] D. Georgakopoulos, M. Hornick, P. Krychniak, and F. Manola. Specification and Management of Extended Transactions in a Programmable Transaction Environment. In *Proceedings of 10th International Conference on Data Engineering*, Houston, Tex., February 1994.

[10] A. K. Elmagarmid, editor, Database Transaction Models for Advanced Applications. Morgan Kaufman, 1991.

[11] Narain Gehani, Krithi Ramamritham, Oded Shmueli. Accessing Extra Database Information: Concurrency Control and Correctness. Computer Science TR 93-081, University of Massachussets, Amherst, 1993.

[12] Jim Gray and Andreas Reuter. Transaction Processing: Concepts and Techniques. Morgan Kaufman, San José, Calif. 1993.

[13] Gregor Kiczales, Jim des Rivières, Daniel G. Bobrow. The Art of the Metaobject Protocol. MIT Press, Cambridge, Mass., 1991.

[14] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, P. Schwartz. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. In *ACM TODS*, 17(1):94–162, 1992.

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

[16] C. Pu, G. Kaiser, G., and N. Hutchinson. Split-Transactions for Open-Ended Activities. In *Proceedings of the 14th International Conference on Very Large Data Bases*, pages 26–37, Los Angeles, CA, Sept. 1988.

[17] Cris Pedregal Martin and Krithi Ramamritham. ARIES/RH: Robust Support for Delegation by Rewriting History. TR 95-51 Computer Science Dept., University of Massachussets, Amherst, June 1995.

[18] Cris Pedregal Martin and Krithi Ramamritham. Delegation: Efficiently Rewriting History. TR 95-90 Computer Science Dept., University of Massachussets, Amherst, October 1995.

[19] Rothermel, K., and C. Mohan. ARIES/NT: A Recovery Method Based on Write-Ahead Logging for Nested Transactions. In *Proceedings of the 15th International Conference on Very Large Databases*, pages 337–346, Amsterdam, 1989.

[20] H. Wächter and A. Reuter. The ConTract Model. In [10].

[21] Gerhard Weikum, Christof Hasse, Peter Broessler, Peter Muth. Multi-Level Recovery. In *ACM International Symposium on Principles of Database Systems*, pages 109–123, Nashville, 1990.