Delegation in ACTA to Control Sharing in Extended Transactions †

Panos K. Chrysanthis
Dept. of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260

Krithi Ramamritham
Dept. of Computer Science
University of Massachusetts
Amherst, MA 01003

Abstract

ACTA is a comprehensive transaction framework that facilitates the formal description of properties of extended transaction models. Specifically, using ACTA, one can specify and reason about (1) the effects of transactions on objects and (2) the interactions between transactions. This paper focuses on one of the building blocks of ACTA, namely delegation. A transaction t_i can delegate *to t_j the responsibility for committing or aborting an operation op. Once this delegation occurs, it is as if t_j performed op and* not *t_i. We discuss how the notion of delegation is useful to capture the interactions that take place in extended transactions.*

1 Introduction

ACTA was introduced in [2] to investigate the formal specification, analysis, and synthesis of extended transaction models. Our goal in this paper is to provide a summary of the notion of *delegation*. Unlike in traditional transactions, sometimes extended transactions require the flexibility of not having to atomically commit all the operations invoked by a transaction. ACTA considers the commitment of an operation as a significant event, in addition to the commitment of a transaction. By separating transaction commitment from operation commitment, ACTA allows for a finer treatment of recovery than is allowed with traditional transactions. Thus, it is possible for a transaction to abort and yet for some of its operations to commit. Furthermore, while a transaction is executing, it may selectively abort some of the operations it has performed and yet commit.

Delegation is a powerful concept and is very useful in the controlled commitment of operations. A transaction t_i can *delegate* to t_j the responsibility for committing or aborting an operation *op*. Once this delegation occurs, it is as if t_j performed *op* and *not* t_i and hence t_j has the responsibility to commit or abort *op*. Consider two transactions t_i and t_j where t_i performs op_i on an object *ob* and then delegates op_i to t_j; t_j then performs op_j on *ob* and then commits. This commitment implies that in the committed state, *ob* reflects the changes done by *both* op_i and op_j even if t_i subsequently aborts. Had t_j so desired, it could have aborted op_i unilaterally.

The fact that t_j is able to observe the changes done by t_i in deciding whether to commit or abort the delegated operation op_i exemplifies how delegation broadens the visibility of a delegatee. Thus, through delegation, a transaction can selectively make tentative or partial results as well as hints, such as, coordination information, accessible to other transactions.

†This material is based upon work supported by the National Science Foundation under grants IRI-9109210 and IRI-9210588 and a grant from University of Pittsburgh.

In the rest of this paper, after introducing the necessary underlying concepts, we give a detailed description of delegation.

2 Events, History, and Properties of Histories

During the course of their execution, transactions invoke transaction management primitives, such as, *Begin, Commit, Spawn* and *Abort*. We refer to these as *significant events*. The semantics of a particular transaction model define the significant events of transactions that adhere to that model. We use ϵ_t to denote the significant event ϵ pertaining to t.

Transactions also invoke operations on objects. We refer to these as *object events* and use $p_t[ob]$ to denote the object event corresponding to the invocation of the operation p on object ob by transaction t.

The concurrent execution of a set of transactions T is represented by H, the *history* [1] of the significant events and the object events invoked by the transactions in the set T. H also indicates the (partial) order in which these events occur. This partial order is consistent with the order of the events of each individual transaction t in T. The predicate $\epsilon \rightarrow \epsilon'$ is true if event ϵ precedes event ϵ' in history H. It is false, otherwise. (Thus, $\epsilon \rightarrow \epsilon'$ implies that $\epsilon \in H$ and $\epsilon' \in H$.)

Each transaction t in execution is associated with a $View_t$ which is the subhistory visible to t at any given point in time. In simplified terms, $View_t$ determines the objects and the *state* of objects visible to t. A view is a projection of the history where the projected events satisfy some predicates, typically on the *current history* H_{ct}. Hence, the partial ordering of events in the view is preserved.

The occurrence of an event in a history can be constrained in one of three ways: (1) An event ϵ can be constrained to occur *only after* another event ϵ'; (2) An event ϵ can occur *only if* a condition c is true; and (3) a condition c can *require* the occurrence of an event ϵ.

Correctness requirements imposed on concurrent transactions executing on a database can be expressed in terms of the properties of the resulting histories. Here are two common types of properties. Further examples can be found at the end of this section. A **Commit-Dependency** of t_j on t_i, specified by $(Commit_{t_j} \in H \Rightarrow (Commit_{t_i} \in H \Rightarrow (Commit_{t_i} \rightarrow Commit_{t_j})))$, says that if both transactions t_i and t_j commit then the commitment of t_i precedes the commitment of t_j. An **Abort-Dependency** of t_j on t_i, specified by $(Abort_{t_i} \in H \Rightarrow Abort_{t_j} \in H)$, states that if t_i aborts then t_j aborts.

Some of the dependencies between transactions arise from their invocation of conflicting operations. Two operations p and q *conflict* in some object state, denoted by *conflict*(p, q), iff their effects on the state of the object or their return values are not independent of their execution order. For instance, suppose operation q is an observer of an object's state and p is a (pure) modifier of the state, i.e., p does not observe the state. (For a simple example, consider an object on which read and write are the only operations supported. Read is an observer and write is a modifier.) Suppose p precedes q in a history, i.e., operation q observes the effects of p. Then, if failure atomicity is desired, then all the direct and indirect effects of an aborting transaction must be nullified. So, t_q, the transaction invoking q, has to abort if t_p, the transaction invoking p, aborts, i.e., t_q has an *abort-dependency* on t_p. Suppose instead that q is a modifier and p is an observer or a modifier. Then, operation q makes p's effects/observations obsolete. If the invoking transactions must be executed serializably, t_q has to be serialized after t_p.

Let us discuss serialization orderings precisely. Let $\mathcal{C}$ be a binary relation on transactions: $(t_i \ \mathcal{C} \ t_j)$ if t_i must be serialized before t_j. Thus, $(t_i \ \mathcal{C} \ t_j)$ if $\exists ob \ \exists p, q \ (conflict(p_{t_i}[ob], q_{t_j}[ob]) \wedge (p_{t_i}[ob] \rightarrow q_{t_j}[ob]))$. Clearly, for a history to be serializable, the $\mathcal{C}$ relation must be acyclic, i.e., $\nexists t \ (t \ \mathcal{C}^* \ t)$ where $\mathcal{C}^*$ is the transitive closure of $\mathcal{C}$.

Depending on the semantics of a transaction and its relationship to others, not all conflicts need produce abort dependencies or serialization orderings. To capture this, with each transaction t in progress ACTA associates a $conflictset_t$, the conflict set of transaction t, to denote those operations

in the current history against which conflicts have to be considered when t invokes an operation. The conflict set is thus a subset of the operation events in the current history, where the events in the subset satisfy predicates, again on H_{ct}.

Other dependencies between transactions arise from the constraints imposed on the manner in which extended transactions are required to be structured. For instance, in the nested transaction model [5], a (parent) transaction *spawns* (child) transactions. A parent can commit only after its children have committed. That is, a parent has a commit dependency on its children. But, if the parent aborts, *and a child has not yet committed*, the child is aborted. In this case, because of the (italicized) qualification associated with this abort-dependency, we say that the child has a *weak abort-dependency* on its parent. The commit dependency of the parent on the child and a weak abort-dependency of the child on the parent are formed when a parent spawns a child.

3 Delegation

A transaction t_i delegates to t_j the responsibility for committing or aborting an operation p when it invokes the event $Delegate_{t_i}[t_j, p]$. Once this delegation occurs, it is as if t_j performed p and *not* t_i. Hence, as a side effect, the dependencies induced by operations performed on the delegated objects are redirected from the delegator to the delegatee. $Delegate_{t_i}[t_j, ops]$ delegates the set of operations *ops* from t_i to t_j.

Via nested transactions, let us illustrate a simple use of delegation. Inheritance in nested transactions is an instance of delegation. Delegation from a child t_c to its parent t_p occurs when t_c commits. This is captured by the following requirement ($\mathsf{Commit}_{t_c} \in H \Leftrightarrow Delegate_{t_c}[t_p, AccessSet_{t_c}] \in H$) where $AccessSet_t$ contains all the operations t is responsible for. That is, all the operations that a child transaction is responsible for are delegated when it commits.

Given the concept of delegation, it is no longer the case that the transaction invoking an operation is the same as the transaction that is responsible for committing (or aborting) the operation. Specifically, once t_i delegates p to t_j, t_j becomes the *responsible transaction* for p, or simply, t_j is responsible for p. This is denoted by $ResponsibleTr(p)$. Note that $ResponsibleTr(p)$ can change as delegations occur and so given H_{ct}, for each operation p in it, there exists a $ResponsibleTr$. If p was never delegated, this transaction is the same as the one that invoked p. Otherwise, this is the transaction to which p was most recently delegated.

Delegation has the following ramifications which are formally stated in [3]:

- $ResponsibleTr(p_{t_i}[ob])$ is t_i, the event-invoker, unless t_i delegates $p_{t_i}[ob]$ to another transaction, say t_j, at which point $ResponsibleTr(p_{t_i}[ob])$ will become t_j. If subsequently t_j delegates $p_{t_i}[ob]$ to another transaction, say t_k, $ResponsibleTr(p_{t_i}[ob])$ becomes t_k.

- The precondition for the event $Delegate_{t_j}[t_k, p_{t_i}[ob]]$ is that $ResponsibleTr(p_{t_i}[ob])$ is t_j. The postcondition will imply that $ResponsibleTr(p_{t_i}[ob])$ is t_k.

- A precondition for the event $Abort_{t_j}[p_{t_i}[ob]]$ (as well as for $Commit_{t_j}[p_{t_i}[ob]]$) is that $ResponsibleTr(p_{t_i}[ob])$ is t_j.

- Delegation cannot occur in case the delegatee has already committed or aborted, and it has no affect if the delegated operations have already been committed or aborted.

- Since once an operation is delegated it is as though the delegatee performed the operation. Thus, delegation (1) brings the delegated operations into delegatee's view if they were not already, and (2) redirects the dependencies induced by delegated operations from the delegator to the delegatee — dependencies are sort of responsibilities.

Delegation can be used not only in controlling the visibility of objects, but also to specify the recovery properties of a transaction model. For instance, if a subset of the effects of a transaction should not be obliterated when the transaction aborts while at the same time they should not be made permanent, the Abort event associated with the transaction can be defined to delegate these effects to the appropriate transaction. In this way, the effects of the delegated operations performed by the delegator on objects are not lost even if the delegator aborts. Instead, the delegatee has the responsibility for committing or aborting these operations. Similarly, as the example of nested transactions illustrated above, by means of delegation, it is possible for a subset of the effects of committed transactions not to be made permanent. This is the simplest method for structuring non-compensatable components of extended transactions, e.g., open-nested transactions [4].

A transaction can delegate at any point during its execution, not just when it aborts or commits. For instance, in Split Transactions [6], a transaction may *split* into two transactions, a splitting and a split transaction, at any point during its execution. A splitting transaction t_a may delegate to the split transaction t_b some of its operations at the time of the split ($\mathsf{Split}_{t_a}[t_b] \in H \Leftrightarrow Delegate_{t_a}[t_b, DelegateSet] \in H$). Here, it is interesting to note that a split transaction can affect objects in the database by committing and aborting the delegated operations even without invoking any operation on them.

Other transaction models using delegation include Reporting Transactions and Co-Transactions described in [3]. A reporting transaction periodically reports to other transactions by delegating its current results. This supports the construction of data-driven computations, e.g., pipeline-like or star-like computations. Co-transactions behave like *co-routines* in which control is passed from one transaction to the another transaction at the time of the delegation. As in the case of reporting transactions, a transaction t_a delegates its current results, contained in $ReportSet_{t_a}$, to its co-transaction t_b by invoking the Join event ($\mathsf{Join}_{t_a}[t_b] \in H \Leftrightarrow Delegate_{t_a}[t_b, ReportSet_{t_a}] \in H$).

In cooperative environments, transactions cooperate by having intersecting views, by allowing the effects of their operations to be visible without producing conflicts, and by delegating responsibilities to each other. By being able to capture these aspects of transactions, the ACTA framework is applicable to cooperative environments.

References

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

[2] Chrysanthis, P. K. and Ramamritham, K. ACTA: A Framework for Specifying and Reasoning about Transaction Structure and Behavior. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 194–203, Atlantic City, NJ, May 1990.

[3] Chrysanthis, P. K. *ACTA, A Framework for Modeling and Reasoning about Extended Transactions*. PhD thesis, University of Massachusetts, Amherst, MA, September 1991.

[4] Chrysanthis, P. K. and Ramamritham, K. Synthesis of Extended Transaction Models using ACTA. *Submitted for publication*, April 1992.

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

[6] Pu, C., Kaiser, G., and Hutchinson, N. Split-Transactions for Open-Ended Activities. In *Proceedings of the Fourteenth International Conference on VLDB*, pages 26–37, Los Angeles, CA, Sept. 1988.