A Formalism for Extended Transaction Models

Panos K. Chrysanthis Krithi Ramamritham
Dept. of Computer and Information Science
University of Massachusetts
Amherst, MA 01003

Abstract

Several extensions to the transaction model adopted in traditional database systems have been proposed in order to support the *functional* and *performance* requirements of emerging advanced applications such as design environments. In [6], we introduced a comprehensive transaction framework, called ACTA to specify the effects of extended transactions on each other and on objects in the database, and to reason about the properties of extended transactions. This paper presents the formalism, underlying ACTA, necessary to prove the visibility, consistency, recovery, and permanence properties of transactions in the extended models. In this paper we show how the formalism can be used to specify and reason about the properties of traditional, nested, and split transaction models.

1 Introduction

Transactions in database systems combine several important notions such as: *visibility*, referring to the ability of one transaction to see the results of another transaction *while* it is executing; *consistency*, referring to the correctness of the state of the database that a committed transaction produces; *recovery*, referring to the ability, in the event of failure, to take the database to some state that is considered correct; and *permanence*, referring to the ability of a transaction to record its results in the database. The flexibility of a given transaction model depends on the way these four notions are combined.

Although powerful, the transaction model [12, 16] found in traditional database systems is found lacking in *functionality* and *performance* when used for applications that involve reactive (endless), open-ended (long-lived) and collaborative (interactive) activities. Hence, various extensions to the traditional model have been proposed, referred to herein as *extended transactions* [23, 28, 2, 25, 19, 15, 27, 18, 10, 4, 11]. Compared to the traditional transaction model, these models associate "broader" interpretations with the four transaction notions mentioned above to provide enhanced functionality while increasing the potential for improved performance. Upon examining these *ad hoc* extensions to the traditional transaction model, one is prompted to seek answers to the following questions:

- What properties does a model possess *vis. a vis.* visibility, consistency, recovery, and permanence? (For e.g., traditional transactions guarantee failure atomicity, serializability, and durability.) What added functionality does a model provide?
- In what respects is a model similar to traditional transactions? In what respects is it dissimilar? More generally, how does one transaction model differ from another? Can two models be used in conjunction?

In attempting to answer these questions, we found a need for a common framework within which one can specify and reason about the nature of interactions between transactions in a particular model. This motivated the development of a comprehensive transaction framework, called *ACTA*[1], which characterizes the effects of transactions as per the taxonomy of Figure 1.

In this paper, we provide the formal underpinnings of ACTA the use of which one can answer the questions posed above. Specifically, the formalism allows the specification of (1) the interactions between transactions in terms of relationships between significant (transaction management) events, such as *begin*, *commit*, *abort*, *delegate*, *split*, and *join*, pertaining to different transactions and (2) transactions' effects on objects' state and concurrency status (i.e., synchronization state).

Section 2 presents some of the preliminary formalism needed to describe the various facets of ACTA. ACTA itself forms the focus of Section 3. Examples of the application of ACTA to (extended) transaction models are provided in Section 4.

2 Preliminaries

2.1 Objects, their state, and status

A database, denoted by DB, is the entity that maintains all the shared objects in a system. A transaction accesses and manipulates the objects in the database by invoking operations specific to individual objects. Each object is characterized by its state *and* status. The *state* of an object is represented by its contents. Each object has a type, which defines a set of operations that provide the only means to create, change

This material is based upon work supported by the N.S.F. under grant CDA 8922572.

First author's present address: Department of Computer Science, 325 Alumni Hall, University of Pittsburgh, PA 15260.

[1]ACTA means *actions* in Latin.

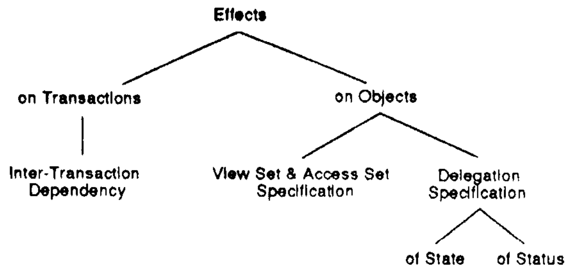

Figure 1: Dimensions of the ACTA framework

and examine the state of an object of that type. It is assumed that an operation always produces an output (return value), that is, it has an outcome (condition code) or a result. The result of an operation on an object depends on the current state of the object. For a given state s of an object, we use $return(s,p)$ to denote the output produced by operation p, and $state(s,p)$ to denote the state produced after the execution of p. The synchronization *status* of the object determines the operations that can be invoked in its current state.

Definition 1: Invocation of an operation on an object is termed an *object event*. The type of an object defines the object events that pertain to it. We use $p_t[ob]$ to denote the object event corresponding to the invocation of the operation p on object ob by transaction t, and OE_t to denote the set of object events that can be invoked by transaction t (i.e., $p_t[ob] \in OE_t$).

The effects of an operation p invoked by a transaction t on an object ob are made permanent in the database when the event $Commit_t[ob.p]$ occurs. Similarly, the effects of an operation p invoked by a transaction t on an object ob are obliterated when the event $Abort_t[ob.p]$ occurs. These operations are defined on every object. Invoked operations that have neither committed nor aborted are termed *ongoing* operations. Typically, an operation is committed only if the invoking transaction commits and it is aborted only if the invoking transaction aborts. In general, it is conceivable that an extended transaction may commit only a subset of its operations on an object while aborting the rest.

2.2 Significant Events and Object Events

In addition to the invocation of operations on objects, transactions invoke transaction management primitives. For example, atomic transactions are associated with three transaction management primitives: *Begin*, *Commit* and *Abort*. The specific primitives and their semantics depend on the specifics of a transaction model. For instance, whereas the *Commit* by an atomic transaction implies that it is terminating successfully and that all of its effects on the objects should be made permanent in the database, the *Commit* of a subtransaction of a nested transaction implies that all of its effects on the objects should be made persistent and visible with respect to its parent and sibling subtransactions[2]. Other transaction management primitives include *Spawn*, found in the nested transaction model, *Split*, found in the split transaction model [25], and *Join*, a transaction termination event found in the split transaction model.

Definition 2: Invocation of a transaction management primitive is termed a ***significant event***. A transaction model defines the significant events that transactions adhering to that model can invoke. SE_t denotes the set of significant events that can be invoked by transaction t.

The set of events E_t invoked by a transaction t is a partial order with ordering relation $<_t$ where $E_t \subseteq (OE_t \cup SE_t)$; i.e., events associated with a transaction are either object events or significant events allowed to be invoked by t, where $<_t$ denotes the temporal order in which the related events are invoked.

2.2.1 Histories and Conditions on Event Occurrences

The concurrent execution of a set of transactions T is represented by the ***history*** [3] containing the object events and significant events invoked by the transactions in the set T and indicates the (partial) order in which these events occur. The partial order of the operations in a history is consistent with the partial order E_t of the events of each individual transaction t in T. We will find it useful to define the ***projection*** of a history H according to a given criterion $\wp$, denoted $P(H,\wp)$. For instance, $P(H,t)$, the projection of a history H on a specific transaction t yields the events associated by t, i.e., $P(H,t) = E_t$; whereas $P(H,ob)$, the projection of the history H on a specific object ob yields the history of operation invocations on the object, i.e., $P(H,ob) = H_{ob}$. A history H_{ob} of an object ob, $H_{ob} = f_1 \circ f_2 \circ \ldots \circ f_n$, indicates both the order of execution of the operations, (f_i precedes f_{i+1}), as well as the functional composition of operations. Thus, a state s of an object produced by a sequence of operations equals the state produced by applying the history H_{ob} corresponding to the sequence of operations on the object's initial state s_0 ($s = state(s_0, H_{ob})$). For brevity, we will use H_{ob} to denote the state of an object produced by H_{ob}, implicitly assuming initial state s_0.

The correctness properties of different transaction models can be expressed in terms of the properties of the histories produced by these models. 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.

Definition 3: 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 $\epsilon' \in H$.)

Definition 4: $(\epsilon \in H) \Rightarrow Condition_H$ specifies that the event ϵ can belong to history H *only*

[2]As shown in Section 3.2, in ACTA, the implications, for objects, of the transaction management operations, are specified via Access and View sets of transactions.

if $Condition_H$ is satisfied. In other words, $Condition_H$ is *necessary* for ϵ to be in H. $Condition_H$ is a predicate involving the events in H.

Consider $(\epsilon' \in H) \Rightarrow (\epsilon \rightarrow \epsilon')$. This states that the event ϵ' can belong to the history H *only if* event ϵ occurs before ϵ'.

Definition 5: $Condition_H \Rightarrow (\epsilon \in H)$ specifies that if $Condition_H$ holds, ϵ should be in the history H. In other words, $Condition_H$ is *sufficient* for ϵ to be in H.

3 The formal ACTA Framework

As was mentioned earlier, ACTA allows the specification of the effects of transactions on other transactions and also their effect on objects. Inter-transaction dependencies, discussed in the next subsection, form the basis for the former while visibility of operations on objects, discussed in Subsection 3.2, form the basis for the latter.

3.1 Effects of Transactions on other Transactions

Dependencies provide a convenient way for specifying and reasoning about the behavior of concurrent transactions and can be precisely expressed in terms of the significant events associated with the transactions. After formally specifying different types of dependencies, we identify the source of these dependencies.

3.1.1 Types of dependencies

Let t_i and t_j be two extended transactions:

Commit-Dependency $(t_j\ \mathcal{CD}\ t_i)$: if both transactions t_i and t_j commit then the commitment of t_i must precede the commitment of t_j; i.e., $(Commit_{t_j} \in H) \Rightarrow ((Commit_{t_i} \in H) \Rightarrow (Commit_{t_i} \rightarrow Commit_{t_j}))$.

Abort-Dependency $(t_j\ \mathcal{AD}\ t_i)$: if t_i aborts, then t_j must abort; i.e., $(Abort_{t_i} \in H) \Rightarrow (Abort_{t_j} \in H)$.

Weak-Abort-Dependency $(t_j\ \mathcal{WD}\ t_i)$: if t_i aborts and t_j has not yet committed, then t_j should also abort. In other words, if t_j commits and t_i aborts then the commitment of t_j should precede the abortion of t_i in a history; i.e., $((Abort_{t_i} \in H) \wedge \neg(Commit_{t_j} \rightarrow Abort_{t_i})) \Rightarrow (Abort_{t_j} \in H)$.

Termination-Dependency $(t_j\ \mathcal{TD}\ t_i)$: t_j cannot commit or abort until t_i either commits or aborts; i.e., $(\epsilon' \in H) \Rightarrow (\epsilon \rightarrow \epsilon')$ where $\epsilon \in \{Commit_{t_i}, Abort_{t_i}\}$, and $\epsilon' \in \{Commit_{t_j}, Abort_{t_j}\}$.

Exclusion-Dependency $(t_j\ \mathcal{ED}\ t_i)$: if t_i commits, then t_j must abort (both t_i and t_j cannot commit); i.e., $(Commit_{t_i} \in H) \Rightarrow (Abort_{t_j} \in H))$.

Compensation-Dependency $(t_j\ \mathcal{CMD}\ t_i)$: if t_i aborts, t_j must commit; i.e., $(Abort_{t_i} \in H) \Rightarrow (Commit_{t_j} \in H)$.

Begin-Dependency $(t_j\ \mathcal{BD}\ t_i)$: transaction t_j cannot begin executing until transaction t_i has begun; i.e., $(Begin_{t_j} \in H) \Rightarrow (Begin_{t_i} \rightarrow Begin_{t_j})$.

Serial-Dependency $(t_j\ \mathcal{SD}\ t_i)$: transaction t_j cannot begin execution until t_i either commits or aborts; i.e., $(Begin_{t_j} \in H) \Rightarrow (\epsilon \rightarrow Begin_{t_j})$ where $\epsilon \in \{Commit_{t_i}, Abort_{t_i}\}$.

The formal definitions of weak-abort-dependency and abort-dependency clearly reflect that weak-abort-dependency is weaker than abort-dependency. Weak-abort-dependency is useful, for e.g., in specifying and reasoning about the properties of nested transactions [23]. Serial-dependency and exclusion-dependency are useful for compensating transactions [19] and contingency transactions [4]. The important difference between exclusion-dependency and compensation-dependency is that exclusion-dependency allows both transactions to abort whereas compensation-dependency does not.

We would like to note that this list of dependencies is *not* exhaustive. Other dependencies that involve significant events besides the Begin, Commit and Abort events, can be defined. As new significant events are associated with extended transactions, new dependencies may be specified in a similar manner. In this sense, ACTA is an open-ended framework.

3.1.2 Source of dependencies

Dependencies between transactions may be a direct result of the structural properties of transactions, or may indirectly develop as a result of interactions of transactions over shared objects. These are elaborated below.

Dependencies due to Structure: The structure of an extended transaction defines its component transactions and the relationships between them. Dependencies can express these relationships and thus, can specify the links in the structure. For example, in hierarchically-structured nested transactions, the parent/child relationship is expressed by a child transaction t_c having a weak-abort-dependency on its parent t_p $(t_c\ \mathcal{WD}\ t_p)$ and a parent having a commit-dependency on its child $(t_p\ \mathcal{CD}\ t_c)$. The weak-abort-dependency guarantees the abortion of an uncommitted child if its parent aborts. Note that this does not prevent the child from committing and making its effects on objects visible to its parent and siblings. (In nested transactions, when a child transaction commits, its effects are not made permanent in the database, they are just made visible to its parent. See Section 4 for a precise formal definition of nested transactions.) The commit-dependency of the parent on its child is preserved if (1) the parent does not commit before its child terminates, or (2) the child aborts in case its parent commits first, i.e., the child becomes an orphan. The weak-abort-dependency together with the commit-dependency ensures that an orphan, i.e., a child transaction whose parent has terminated, will not commit.

Other hierarchically-structured transactions may define different relationships between a parent and different child transactions. For example, in the transaction model proposed in [4, 14] a parent can commit only if its *vital* children commit, i.e., a parent transaction has an abort-dependency on its *vital* children t_v $(t_p\ \mathcal{AD}\ t_v)$. Child transactions may also have different dependencies with their parents if the transaction model supports various spawning or coupling modes [9]. Sibling transactions may also be interrelated in different ways. For example, components of a

Saga [15] can be paired according to a compensated-for/compensating relationship [19]. Relations between a compensated-for and compensating transactions as well as those between them and the saga can be specified via exclusion-dependency, serial-dependency and compensation-dependency. In a similar fashion dependencies that occur in the presence of alternative transactions and contingency transactions [4] can also be specified.

Dependencies due to Behavior: Dependencies formed by the interactions of transactions over a shared object are determined by the object's synchronization properties. Broadly speaking, two operations conflict if the order of their execution matters. For example, in the traditional framework, a compatibility table [3] of an object *ob* expresses simple relations between conflicting operations. A conflict relation has the form $(p_{t_i}[ob] \rightarrow q_{t_j}[ob]) \Rightarrow (t_j\ \mathcal{D}\ t_i)$, indicating that if transaction t_j invokes an operations p and later a transaction t_j invokes an operation q on the same object *ob*, then t_j should develop a dependency of type $\mathcal{D}$ on t_i. As we will see in the next section, ACTA allows conflict relations to be complex expressions involving different types of dependencies, operation arguments, and results, as well as operations on the same or different objects.

3.2 Objects and the Effects of Transactions on Objects

In order to better understand the effects of transactions on objects, we need to first understand the effects of the operations invoked by the transactions.

3.2.1 Conflicts between operations and the dependencies induced by them

Definition 6: Two operations p and q *conflict* in a state produced by H_{ob}, denoted by *conflict*(H_{ob}, p, q), iff

$$\begin{array}{rcl} (state(H_{ob} \circ p,\ q) & \neq & state(H_{ob} \circ q,\ p)) \quad \vee \\ (return(H_{ob},\ q) & \neq & (return(H_{ob} \circ p,\ q)) \quad \vee \\ (return(H_{ob},\ p) & \neq & (return(H_{ob} \circ q,\ p)) \end{array}$$

Two operations that do not conflict are *compatible*.

(Recall that $\circ$ denotes functional composition; $H \circ p$ appends p to history H.) Thus, two operations conflict if their effects on the state of an object or their return values are not independent of their execution order. Since state changes are observed only via return values, only the return values need to be considered in dealing with conflicting operations.

Definition 7: Given *conflict*(H_{ob}, p, q), *return-value-independent*(H_{ob}, p, q) is true if the return value of q is independent of whether p precedes q, i.e., $return(H_{ob} \circ p, q) = return(H_{ob}, q)$; otherwise q is *return-value-dependent* on p (*return-value-dependent*(H_{ob}, p, q)).

Given a history H in which $p_{t_i}[ob]$ and $q_{t_j}[ob]$ occur, the state of *ob* when p_{t_i} is executed is known from where p_{t_i} occurs in the history. Hence, from now on, we drop the first argument in *conflict*, *return-value-independent*, and *return-value-dependent* when it is implicit from the context.

Serializability requirements induce the following dependencies between transactions invoking conflicting operations.

(1) When an operation q follows operation p and *return-value-dependent*(p, q), the transaction t_j invoking the operation q must abort q if for some reason the transaction t_i aborts p; i.e., $(\textit{return-value-dependent}(p, q) \wedge (p_{t_i}[ob] \rightarrow q_{t_j}[ob])) \Rightarrow ((Abort_{t_i}[ob.p] \in H) \Rightarrow (Abort_{t_j}[ob.q] \in H))$.

(2) When an operation p precedes q and *return-value-independent*(p, q), the transaction t_j that invoked q cannot commit q until the transaction t_i that invoked p commits or aborts p; i.e., $(\textit{conflict}(p,q) \wedge \textit{return-value-independent}(p,q) \wedge (p_{t_i}[ob] \rightarrow q_{t_j}[ob])) \Rightarrow ((Commit_{t_j}[ob.p] \in H) \Rightarrow ((Commit_{t_i}[ob.q] \in H) \Rightarrow (Commit_{t_i}[ob.p] \rightarrow Commit_{t_j}[ob.q])))$.

Motivated by this, in ACTA, the concurrency properties of an object are formally expressed in terms of *conflict relations* of the form: $(p_{t_i}[ob] \rightarrow q_{t_j}[ob]) \Rightarrow Condition_H$, where $Condition_H$ is typically a dependency relationship involving the transactions t_i and t_j invoking conflicting operations p and q on an object *ob*. Obviously, the absence of a conflict relation between two operations defined on an object indicates that the operations are compatible and do not induce any dependency[3].

This generality allows ACTA to encompass both object-specific and transaction-specific semantic information. First consider some object-specific semantics. *Commutativity* does not distinguish between return-value dependent and independent conflicts. It treats both the same and uses abort-dependency for both: $(p_{t_i}[ob] \rightarrow q_{t_j}[ob]) \Rightarrow (t_j\ \mathcal{AD}\ t_i)$. *Recoverability* [1] avoids the unnecessary development of an abort-dependency for return-value independent conflicts. Thus, an operation q which is return-value independent of p where p and q conflict, induces the conflict relation: $(p_{t_i}[ob] \rightarrow q_{t_j}[ob]) \Rightarrow (t_j\ \mathcal{CD}\ t_i)$; whereas an operation q being return-value dependent on p induces the conflict relation: $(p_{t_i}[ob] \rightarrow q_{t_j}[ob]) \Rightarrow (t_j\ \mathcal{AD}\ t_i)$.

We introduce *transaction-specific* semantics through an example. Consider a Page object with the standard *read* and *write* operations, where read and write operations conflict. A read is return-value dependent on write, whereas a write is return-value independent of a read or another write. In addition, consider transactions which have ability to reconcile potential read-write conflicts: When a transaction t_i reads a page x and another transaction t_j subsequently writes x,

[3]Clearly, when an invoked operation conflicts with an operation in progress, a dependency, e.g., an abort or commit dependency, will be formed if the invoked operation is allowed to execute. That is, this may induce an abortion or induce a specific commit ordering. One way to avoid this is to force the invoking transaction to (a) wait till the conflicting operation terminates (this is what the traditional "no" entry in a compatibility table means) or (b) abort. In either case, conflict relationships between operations imply that the transaction management system must keep track of ongoing operations and of dependencies that have been induced by the conflict. A commonly used synchronization mechanism for keeping track of ongoing operations and dependencies is based on (logical) *locks*.

t_i and t_j can commit in any order. However, if t_j commits before t_i commits, t_i must reread x in order to commit. This is captured by the following conflict relation: $(read_{t_i}[x] \rightarrow write_{t_j}[x]) \Rightarrow ((Commit_{t_j} \rightarrow Commit_{t_i}) \Rightarrow (Commit_{t_j}[x] \rightarrow read_{t_i}[x])))$.
This conflict relation cannot be derived solely from the object-specific semantics of the page. Clearly, transaction specific concurrency control might *not* achieve serializability but still preserves consistency.

In the example, t_i has to reread the page x when it is subsequently written and committed by t_j. In general, t_i may need to invoke an operation on the same or different object. For instance, instead of x, t_i may have to read a *scratch-pad* object which t_i and t_j use to determine and reconcile potential conflicts. Thus, ACTA allows the specification of operations that need to be controlled in correct histories as well as operations that have to occur in correct histories. These correspond to *conflicts* and *patterns* in [27].

The $Condition_H$ in a conflict relation may include other significant events defined by the various transaction models. As an example, consider the significant event *Notify*, related to the notion of *notification* useful in a cooperative environment [13]. For instance, the condition $Notify_{t_j}[(t_i\ \mathcal{CD}\ t_j)]$ will cause a commit-dependency to be established from transaction t_i to t_j as well as *notify* t_j about the development of the commit-dependency. Such a pair of conditions can be used to define a recoverability-based table in a cooperative environment. Transaction t_j can use the information about the existence of the commit-dependency to postpone the invocation of another operation that causes a commit-dependency of t_j on t_i, and thus postpone the formation of a circular commit dependency.

The generality of the conflict relations allows ACTA to capture different types of type-specific concurrency control discussed in the literature [26, 17, 1], and even to tailor them for cooperative environments.

3.2.2 Controlling object visibility

As defined earlier, Visibility refers to the ability of one transaction to see the results of another transaction *while* it is executing. In ACTA, visibility of the transactions is captured by associating with every transaction two sets of objects: *View Set* which contains all the objects potentially visible to the transaction, and *Access Set* which contains all the objects already accessed by the transaction. When an object in the View Set of a transaction is accessed by the transaction or a new object is created by the transaction, the object becomes a member of the transaction's Access Set. $AccessSet_t$ refers to the Access Set of a transaction t, and $ViewSet_t$ refers to the View Set of t.

Definition 8: $AccessSet_t = \{ob \mid \exists p(p_t[ob] \in H)\}$; i.e., $AccessSet_t$ contains all the objects upon which t has invoked an operation.

Definition 9: $ViewSet_t = AccessSet_t \cup \{\cup AccessSet_{t_i} \mid t_i \in Tset_t\} \cup DB$ where the constitution of $Tset_t$ is determined by a given transaction model.

The View Set of a transaction is expressed in terms of its AccessSet, other transactions' Access Sets and the database. $Tset_t$ contains the transactions whose Access Sets constitute the View Set of transaction t. That is, $Tset_t$ specifies the composition of the View Set of t. Rules for composing the View Set of a transaction are determined by the specific transaction model.

A transaction t can invoke an operation on an object in $ViewSet_t$ without conflicting with any transactions t_i in $Tset_t$. That is, no dependencies are induced between t and t_i when t invokes an operation q that conflicts with an operation p invoked by t_i where p precedes q. Therefore, with the introduction of the View Set of a transaction, the conflict relations associated with an object need to be redefined. Specifically, when $(p_{t_i} \rightarrow q_{t_j})$, the dependency relations between transactions t_i and t_j specified by $Condition_H$ will be induced only if t_i is not in $Tset_{t_j}$, i.e., $((p_{t_i} \rightarrow q_{t_j}) \wedge \neg(t_i \in Tset_{t_j})) \Rightarrow Condition_H$.

Note that an object in $ViewSet_t$ may occur in more than one of ViewSet's components. For instance, suppose $ob \in AccessSet_{t_1} \wedge ob \in AccessSet_{t_2}$ and $ViewSet_t = AccessSet_{t_1} \cup AccessSet_{t_2} \cup DB$. This implies that both t_1 and t_2 have performed an operation on ob. In order to determine whether t can perform an operation on ob, it will be necessary to determine the status of ob in both $AccessSet_{t_1}$ and $AccessSet_{t_2}$. In general, the constituents of a ViewSet may have to be visited in a certain order to determine conflicts. The order in which these are considered is specified by a relation, called *order of conflict checks* (denoted by $AccessOrder$).

We illustrate the notion of Access Set and View Set by considering nested transactions. In nested transactions, $Tset_t = \{t_{n-1}, t_{n-2}..., t_0\}$, where $t_n = t$, $t_i\ \mathcal{WD}\ t_{i-1}$, $0 < i \leq n$, and t_0 is the root of a nested transaction. The weak-abort-dependency $\mathcal{WD}$ uniquely specifies that t_{i-1} is the parent of t_i. A transaction t can invoke an operation (on an object in its View Set) that conflicts with operations invoked by its ancestors t_a without forming any dependency since t_a is in $Tset_t$. Conflicts are determined with respect to uncommitted operations on an object. In nested transactions, because of serializability, the AccessSets constituting the ViewSet of t do not have to be visited in a certain order to determine conflicts[4]. In general, it may be necessary to visit an object ob which is in different AccessSets, in a particular order in order to determine conflicts since different conflict relations may be associated with ob at different levels in the hierarchy, e.g, as in *multilevel-correctness* [21]. Hence, the need for AccessOrder specification.

In nested transactions, when the root commits, its effects are made permanent in the database, whereas when a subtransaction commits, via delegation, its effects are made visible to its parent transaction.

In general, a transaction may *delegate* the responsibility for finalizing its effects on some of the objects in its Access Set to another transaction. This is achieved by modifying the history of the delegated objects to reflect that operations invoked on them by the first transaction t_i (i.e., the delegator) were invoked by the

[4]Certain implementations of nested transactions, for e.g., the lock-based in [23], require to consider conflicts in a particular order. Specifically, in [23], the order of conflict checks for a subtransaction t_n is defined as: $AccessOrder_{t_n} = (AccessSet_{t_n}, AccessSet_{t_{n-1}}, ...AccessSet_{t_0}, DB)$.

second transaction t_j (i.e., the delegatee). In particular, $(Delegate_{t_i}[t_j] \in H) \Rightarrow (\forall ob \in DelegateSet(t_i, t_j), \forall p\ (p_{t_i}[ob] \rightarrow Delegate_{t_i}[t_j]), replace(p_{t_i}[ob], p_{t_j}[ob]))$[5] This effectively redirects the dependencies induced by operations performed on the delegated objects from the delegator to the delegatee and removes the objects in the *delegateset*(t_i, t_j) from the Access Set of the delegator and adds them to the Access Set of the delegatee. Delegation effectively broadens the visibility of the delegatee and is useful in selectively making tentative or partial results as well as hints, such as, coordination information, accessible to other transactions. Delegation fails in the event that the delegatee has already committed or aborted.

The notion of inheritance used in nested transactions is an instance of delegation. Specifically, when a child transaction t_c commits, t_c delegates to its parent t_p all the objects that it has accessed $(DelegateSet(t_c, t_p) = AccessSet_{t_c}$, where $t_c \mathcal{WD}\ t_p)$. Delegation need not only occur upon commit or abort but a transaction can delegate any of the objects in its Access Set to another transaction at any point during its execution. This is the case for Co-Transactions and Reporting Transactions that we describe in [5].

Delegation can be used not only in controlling the visibility of objects, but it can also be used 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 significant event associated with the transaction can be defined to delegate these effects to the appropriate transaction. In this way, the effects of the delegator on the delegated objects are not obliterated even if the delegator aborts.

In cooperative environments, transactions (components) cooperate by having intersecting Access Sets and View Sets, by delegating objects to each other, or by *notifying* each other of their behavior. By being able to capture these aspects of transactions, the ACTA framework is designed to be applicable to cooperative environments.

4 Examples of Correctness of Extended Transaction Models

ACTA has been successfully used for characterizing the structure and behavior of a number of extended transactions models [6, 5, 4, 22]. Here we show how these characterizations can be used to reason about the correctness properties, e.g., concurrency and recovery properties, of some of these models.

[5]This basically says that once delegation occurs, history is "rewritten" to indicate that any operation p invoked on a delegated object ob by the delegator t_i is instead considered to be invoked by the delegatee t_j. To be more precise we should say that once the event $Delegate_{t_i}[t_j]$ is appended to the history and $ob \in DelegateSet(t_i, t_j)$, all the conflicts and dependencies that t_i is associated with should instead be associated with t_j.

4.1 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.

Let us first define the correctness properties of objects within formal ACTA, starting with the serializability correctness criterion.

Definition 10: Let H be a history. Let C be a binary relation on transactions, and t_i and t_j be transactions, $t_i \neq t_j$. $t_i C t_j$ if $\exists ob\ \exists p, q, (conflict(p, q) \wedge (p_{t_i}[ob] \rightarrow q_{t_j}[ob]))$.

Definition 11: Let C^* be the transitive-closure of C; i.e., $t_i C^* t_k$ if $t_i C t_k$ or $\exists t_j, (t_i C^* t_j \wedge t_j C^* t_k)$.

Definition 12: A set of transactions T is *serializable* iff $\forall t \in T, \neg(t C^* t)$.

Definition 13: An object ob behaves *correctly* iff $\forall t_i, t_j, t_i \neq t_j\ \forall p, q,$
$(return\text{-}value\text{-}dependent(p,q) \wedge (p_{t_i}[ob] \rightarrow q_{t_j}[ob])) \Rightarrow$
$((Abort_{t_i}[ob.p] \in H_{ob}) \Rightarrow (Abort_{t_j}[ob.q] \in H_{ob}))$.

This definition implies that for an object to behave *correctly* it must ensure that when an operation aborts, any return-value dependent operation that follows it must also be aborted. It is not necessary for it to exhibit serial behavior, i.e., it is not necessary for the order in which the operations are executed by different transactions to be serializable.

Definition 14: An object ob behaves *serializably* iff (1) $\forall t_i, t_j, t_i \neq t_j\ \forall p, q, (conflict(p,q) \wedge$
$(p_{t_i}[ob] \rightarrow q_{t_j}[ob])) \Rightarrow ((Commit_{t_j}[ob.q] \in H_{ob}) \Rightarrow$
$((Commit_{t_i}[ob.p] \in H_{ob}) \Rightarrow$
$(Commit_{t_i}[ob.p] \rightarrow Commit_{t_j}[ob.q])))$,
(2) $\forall t \forall p, (Commit_t[ob.p] \in H_{ob}) \Rightarrow \neg(t C^* t)$.

This definition states that the serializable behavior of an object is ensured by defining a commit order between transactions invoking conflicting operations and by preventing transactions from forming cyclic C relationships.

Definition 15: An object ob is *atomic* if ob behaves *correctly* and *serializably*.

Definition 16: Transaction t is *failure atomic* if
1. $\exists ob(\exists q\ Commit_t[ob.q] \in H) \Rightarrow (Commit_t \in H)$;
2. $(Commit_t \in H) \Leftrightarrow$
   $\forall ob \forall q((q_t[ob] \in H) \Rightarrow (Commit_t[ob.q] \in H))$;
3. $\exists ob(\exists q\ Abort_t[ob.q] \in H) \Rightarrow (Abort_t \in H)$; and
4. $(Abort_t \in H) \Rightarrow$
   $\forall ob \forall q((q_t[ob] \in H) \Rightarrow (Abort_t[ob.q] \in H))$.

As mentioned earlier, failure atomicity implies that all or none of a transaction's operations are executed. In the above definition, the "all" clause is captured by condition (1) which states that if an operation is committed on an object, the invoking transaction must commit, and condition (2) which states that a transaction commits iff all the operations invoked by the transaction are committed. The "none" clause is captured by condition (3) which states that if an operation is aborted on an object, the invoking transaction must abort, and condition (4) which states that if a transaction aborts, all the operations invoked by the transaction are aborted.

Now let us express the basic properties of atomic transactions with a set of axioms.

Definition 17:
Axiomatic definition of Atomic Transactions
Let t be an atomic transaction.
1. $ES_t = \{Begin, Commit, Abort\}$.
2. $(Begin_t \in H) \Rightarrow (\neg(Commit_t \rightarrow Begin_t) \wedge \neg(Abort_t \rightarrow Begin_t) \wedge \neg(Begin_t \rightarrow Begin_t))$.
3. $(Begin_t \in H) \Rightarrow ((ViewSet_t = AccessSet_t \cup DB) \wedge (AccessOrder_t = (AccessSet_t, DB)))$.
4. t is *failure atomic.*
5. $(Commit_t \in H) \Rightarrow ((Begin_t \rightarrow Commit_t) \wedge \neg(Abort_t \in H))$.
6. $(Abort_t \in H) \Rightarrow ((Begin_t \rightarrow Abort_t) \wedge \neg(Commit_t \in H))$.
7. $\forall ob, (\exists p, p_t[ob] \in H) \Rightarrow (ob\ is\ atomic)$.
8. $(Commit_t \in H) \Rightarrow \neg(tC^*t)$.

Axiom 1 states that atomic transactions are associated with the three significant events: Begin, Commit and Abort. Axiom 2 states that the $Begin_t$ event which instantiates a new atomic transaction can be invoked at most once by a transaction. Axiom 3 restricts the visibility of a transaction to the objects in its Access Set and the database. Axiom 4 expresses the failure atomicity property of atomic transactions. Axioms 5 states that only an instantiated transaction can commit and that an atomic transaction cannot be committed after it has been aborted. Similarly, Axiom 6 states that only an instantiated transaction can abort and an atomic transaction cannot be aborted after it has been committed. Axiom 7 specifies that all objects upon which an atomic transaction invokes an operation are atomic objects. That is, they detect conflicts and induce the appropriate dependencies. Finally, Axiom 8 states that an atomic transaction can commit only if it is not part of a cycle of C relations. Note that the atomicity property local to individual objects is not sufficient to guarantee serializable execution of concurrent transactions across all objects [29].

Definition 18: An atomic transaction management scheme is *correct* if it conforms to definition 17.

4.2 Nested Transactions

In the Nested Transaction model, e.g. [23], transactions are composed of subtransactions or child transactions designed to localize failures within a transaction and to exploit parallelism within transactions. A subtransaction can be further decomposed into other subtransactions, and thus, a transaction may expand in a hierarchical manner. Subtransactions execute atomically with respect to their siblings, are failure atomic with respect to their parent, and can abort independently without causing the abortion of the whole transaction.

A subtransaction can potentially access any object that is currently accessed by one of its ancestor transactions. In addition, any object in DB is also potentially accessible to the subtransaction. 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 in DB only when the root transaction commits.

The nested transaction model supports two types of transactions, namely, *root transactions* and *subtransactions*, which are associated with different significant events (Axioms 1 and 2). The semantics of root transactions are similar to atomic transactions (Axioms 3–8 and 18). The *Abort* event has the same semantics for both transaction types which are the same as the Abort in atomic transactions (Axioms 6–7 and 14–15). However, the semantics of the *Commit* event are different for each transaction type. In the case of a root transaction, Commit has the semantics of the Commit event in atomic transactions (Axioms 5, 7 and 8). In contrast, when a subtransaction commits, through delegation, the objects in its Access Set are made persistent and visible only to its parent transaction (Axiom 13).

Spawn is used to instantiate a new subtransaction. The spawn event establishes a parent/child relationship between the spawning and spawned transactions. This relationship is reflected by the weak-abort-dependency and commit dependency between the related transactions (Axiom 10). The ability of a subtransaction to access any object currently accessed by one of its ancestor transactions is expressed by defining the View Set of the subtransaction in terms of the Access Sets of its ancestor transactions (Axiom 11).

Definition 19:
Axiomatic definition of Nested Transactions
Let t_0 be the root transaction, t_p be a root or a subtransaction, and t_c be a subtransaction of t_p.
1. $ES_{t_0} = \{Begin, Spawn, Commit, Abort\}$.
2. $ES_{t_c} = \{Spawn, Commit, Abort\}$.
3. $(Begin_{t_0} \in H) \Rightarrow (\neg(Commit_{t_0} \rightarrow Begin_{t_0}) \wedge \neg(Abort_{t_0} \rightarrow Begin_{t_0}) \wedge \neg(Begin_{t_0} \rightarrow Begin_{t_0}))$.
4. $(Begin_{t_0} \in H) \Rightarrow ((ViewSet_{t_0} = AccessSet_{t_0} \cup DB) \wedge (AccessOrder_{t_0} = (AccessSet_{t_0}, DB)))$.
5. $(Commit_{t_0} \in H) \Rightarrow ((Begin_{t_0} \rightarrow Commit_{t_0}) \wedge \neg(Abort_{t_0} \rightarrow Commit_{t_0}))$.
6. $(Abort_{t_0} \in H) \Rightarrow ((Begin_{t_0} \rightarrow Abort_{t_0}) \wedge \neg(Commit_{t_0} \in H))$.
7. t_0 is *failure atomic.*
8. $(Commit_{t_0} \in H) \Rightarrow \neg(t_0C^*t_0)$.
9. $(Spawn_{t_p}[t_c] \in H) \Rightarrow (\neg(Commit_{t_p} \rightarrow Spawn_{t_p}[t_c]) \wedge \neg(Abort_{t_p} \rightarrow Spawn_{t_p}[t_c]))$.
10. $(Spawn_{t_p}[t_c] \in H) \Leftrightarrow ((t_c\ \mathcal{WD}\ t_p) \wedge (t_p\ \mathcal{CD}\ t_c))$.
11. $(Spawn_{t_p}[t_c] \in H) \Rightarrow ((ViewSet_{t_c} = \{\cup AccessSet_{t_i} | t_i \in Tset_{t_c}\} \cup DB) \wedge (Tset_{t_c} = \{t_n, t_{n-1}, t_{n-2} \ldots, t_0\}))$,
    where $t_n = t_c$, $t_i\ \mathcal{WD}\ t_{i-1}$.
12. $(Commit_{t_c} \in H) \Rightarrow ((Spawn_{t_p}[t_c] \rightarrow Commit_{t_c}) \wedge \neg(Abort_{t_p}[t_c] \rightarrow Commit_{t_c}))$.
13. $((Commit_{t_c} \in H) \Leftrightarrow (Delegate_{t_c}[t_p] \in H)) \wedge (DelegateSet(t_c, t_p) = AccessSet_{t_c})$.
14. $(Abort_{t_c} \in H) \Rightarrow ((Spawn_{t_p}[t_c] \rightarrow Abort_{t_c}) \wedge \neg(Commit_{t_c} \rightarrow Abort_{t_c}))$.
15. $\exists ob(\exists q\ Abort_{t_c}[ob.q] \in H) \Rightarrow (Abort_{t_c} \in H)$.
16. $(Abort_{t_c} \in H) \Rightarrow \forall ob \forall q, (((q_t[ob] \in H) \Rightarrow (Abort_{t_c}[ob.q] \in H))$.
17. $\forall t_a, t, t\ \mathcal{WD}^*t_a\ \forall ob\ \forall p, (p_t[ob] \in H) \Rightarrow \nexists q((p_t[ob] \rightarrow q_{t_a}[ob]) \wedge conflict(p,q))$.
18. $\forall t, t = t_0 \vee t = t_c\ \forall ob$

$(\exists p, p_t[ob] \in H) \Rightarrow (ob \text{ is atomic})$.
19. $(Commit_{t_c} \in H) \Rightarrow \neg(t_c C^* t_c)$.

Axiom 17 states that given transaction t and its ancestor t_a and conflicting operations p and q, t_a cannot invoke q after t invokes p. In the absence of this restriction, it would be possible for an ancestor t_a of a transaction t to develop an abort-dependency on t (t_a $\mathcal{AD}$ t) by invoking an operation that is return-value dependent on a preceding operation invoked by t. In such a case in which a parent transaction develops an abort-dependency on its child, if the child aborts, the parent also aborts. This means that it would be possible for a subtransaction to cause the abortion of its parent and possibly of the whole nested transaction (if the parent happens to be the root transaction). But this violates the property of nested transactions that localizes failures by allowing a subtransaction to abort independently without causing the abortion of the whole transaction.

Although Axioms 7, 8, 18 and 19 would be sufficient to ensure the serializability of atomic transactions, they are not in the case of nested transactions because of Axioms 11 and 13 which allow dependencies between two transactions to be ignored or redirected.

Based on the above axiomatic definition of nested transactions, the failure semantics and the serializability property of nested transactions can be shown.

Lemma 1: *No Orphan Commits Lemma*
Let H be a history of a nested transaction, t_p and t_c be transactions where t_p be the parent of t_c.
$(((Commit_{t_p} \in H) \wedge \neg(Commit_{t_c} \rightarrow Commit_{t_p})) \vee$
$((Abort_{t_p} \in H) \wedge \neg(Commit_{t_c} \rightarrow Abort_{t_p}))) \Rightarrow$
$(Abort_{t_c} \in H)$.

Informally, this states that an orphan, i.e., a child whose parent has either committed or aborted before it has terminated, will be aborted. This lemma is derived from (t_p $\mathcal{CD}$ t_c) and (t_c $\mathcal{WD}$ t_p) which are a result of Axiom 10.

Definition 20: Let $C(H)$ be the committed projection of a history H.
$C(H) = P(H, \{t | Commit_t \in H\})$.

Theorem 1: A nested transaction t_0 has the following properties:
(1) $(Abort_{t_0} \in H) \Rightarrow \forall t, t\ \mathcal{WD}^* t_0\ \forall ob\ \forall p$,
$((q_t[ob] \in H) \Rightarrow (Abort_t[ob.p] \in H))$;
(2) $\exists t\ \exists p, (p_t \in C(H)) \Rightarrow (t \text{ is a root transaction})$;
(3) if H a history of nested transactions, $C(H)$ is *serializable*.

Property (1) which says that if a nested transaction aborts, the operations invoked by its root and its subtransactions are all aborted, follows from the *no orphan commits* lemma and Axioms 7, 15 and 16. Properties (2) and (3) require that the effects of operations of root transactions are committed in the database in a serializable fashion. These properties are derivable from the semantics of delegation (note that once delegation is performed by a subtransaction when it commits, the (committed) subtransaction in $C(H)$ is not associated with any operations and that delegation preserves conflicts in a history), the semantics of atomic objects, the *no orphan commits* lemma, and axiom 8.

Definition 21: A nested transaction management scheme is *correct* if it conforms to definition 19.

4.3 Split Transactions

In the Split Transaction model [25], it is possible for a transaction t_a to split into two transactions, t_a and t_b. t_a and t_b transactions may be *independent*, in which case they can commit or abort independently, or they may be *serial*, in which case t_a must commit in order for t_b to commit. Whether t_a and t_b transactions are independent or serial depends on the objects accessed by them.

In the split transaction model, a transaction can be instantiated through either the *Begin* significant event, called *primary* transaction, or the *Split* significant event, called *split* transaction. Although primary and split transactions are associated with different significant events (Axioms 1 and 2), the events with the same name share the same semantics (Axioms 5–15). In fact, the *Begin, Commit* and *Abort* events have the same semantics as the corresponding events in atomic transactions.

The $Split_{t_a}[t_b]$ event splits a transaction into a *splitting* transaction t_a and *split* transaction t_b. Note that a split transaction t_b can invoke $Split_{t_b}[t_c]$ to create another split transaction t_c. In addition, a transaction t_a can invoke $Split_{t_a}[t_c]$ after invoking $Split_{t_a}[t_b]$. This leads to hierarchically structured transactions. For simplicity, we confine our attention in the rest of this Section to the situation when a transaction invokes the split event at most once.

In contrast to the transaction instantiated by the Begin event, through delegation, split transactions may be associated with a non-empty Access Set (Axiom 10). The ability of a split transaction to access specific objects accessed by its splitting transaction is expressed by defining the View Set of the split transaction in terms of a subset of the Access Set ($CanAccess(t_a, t_b)$) of the splitting transaction (Axiom 11). $CanAccess(t_a, t_b)$ contains the objects that t_a has accessed up to the split, and t_b can potentially access after the split. A splitting transaction cannot invoke an operation on an object that conflicts with operations of its split transactions (and their splits) (Axiom 13). After the split t_a may still invoke an operation on an object in $CanAccess(t_a, t_b)$ as long as the operation does not conflict with an operation invoked by t_b. A split is independent, if $CanAccess(t_a, t_b)$ is empty. In the case of *serial split* in which $CanAccess(t_a, t_b)$ is not empty, t_b develops an abort-dependency on t_a[6] (Axiom 14).

Definition 22:
Axiomatic definition of Split Transactions
Let t_p be a primary transaction, t_a be a splitting transaction, t_b be a split of t_a.
1. $ES_{t_p} = \{Begin, Split, Commit, Abort\}$.
2. $ES_{t_b} = \{Split, Commit, Abort\}$.
3. $(Begin_{t_p} \in H) \Rightarrow (\neg(Commit_{t_p} \rightarrow Begin_{t_p}) \wedge$
$\neg(Abort_{t_p} \rightarrow Begin_{t_p}) \wedge \neg(Begin_{t_p} \rightarrow Begin_{t_p}))$.

[6] By taking into consideration the semantics of operations on the individual objects in $CanAccess(t_a, t_b)$, it would be possible to induce weaker dependencies, e.g. commit-dependency, rather than abort-dependency.

4. $(Begin_{t_p} \in H) \Rightarrow ((ViewSet_{t_p} = AccessSet_{t_p} \cup DB) \wedge (AccessOrder_{t_p} = (AccessSet_{t_p}, DB)))$.
5. $(Split_{t_a}[t_b] \in H) \Rightarrow (\neg(Commit_{t_a} \rightarrow Split_{t_a}[t_b]) \wedge \neg(Abort_{t_a} \rightarrow Split_{t_a}[t_b]))$.
6. $(Commit_{t_p} \in H) \Rightarrow ((Begin_{t_p} \rightarrow Commit_{t_p}) \wedge \neg(Abort_{t_b} \rightarrow Commit_{t_b}))$.
7. $(Abort_{t_p} \in H) \Rightarrow ((Begin_{t_p} \rightarrow Abort_{t_p}) \wedge \neg(Abort_{t_p} \rightarrow Abort_{t_p}))$.
8. $(Commit_{t_b} \in H) \Rightarrow ((Split_{t_a}[t_b] \rightarrow Commit_{t_b}) \wedge \neg(Abort_{t_b} \rightarrow Commit_{t_b}))$.
9. $(Abort_{t_b} \in H) \Rightarrow ((Split_{t_a}[t_b] \rightarrow Abort_{t_b})) \wedge \neg(Abort_{t_b} \rightarrow Abort_{t_b}))$.
10. $((Split_{t_a}[t_b] \in H) \Leftrightarrow (Delegate_{t_a}[t_b] \in H)) \wedge (DelegateSet(t_a, t_b) \subset AccessSet_{t_a})$.
11. $(Split_{t_a}[t_b] \in H) \Rightarrow ((ViewSet_{t_b} = AccessSet_{t_b} \cup CanAccess(t_a, t_b) \cup DB) \wedge (AccessOrder_{t_b} = (AccessSet_{t_b}, CanAccess(t_a, t_b), DB)))$.
12. $(CanAccess(t_a, t_b) \subset AccessSet_{t_a}) \wedge (CanAccess(t_a, t_b) \cap DelegateSet(t_a, t_b) = \phi)$.
13. $\forall t_a, t, t\mathcal{AD}^* t_a \; \forall ob \; \forall p, (p_t[ob] \in H) \Rightarrow \nexists q((p_t[ob] \rightarrow q_{t_a}[ob]) \wedge conflict(p,q))$.
14. $(CanAccess(t_a, t_b) \neq \phi) \Rightarrow (t_b \mathcal{AD} t_a)$.
15. t is *failure atomic*.
16. $\forall t, t = t_p \vee t = t_b \; \forall ob \; \forall p, (p_t[ob] \in H) \Rightarrow (ob \textit{ is atomic})$.
17. $\forall t, t = t_p \vee t = t_b, (Commit_t \in H) \Rightarrow \neg(t\mathcal{C}^* t)$.

As in the case of nested transactions, Axioms 16 and 17 are not sufficient to ensure serializability of split transactions due to Axioms 10 and 11. However, split transactions are serializable as we show below.

Lemma 2: A primary transaction t_p is an *atomic transaction* if it does not split.

Lemma 3: Let t_a be the splitting transaction and t_b be the split transaction. If $CanAccess(t_a, t_b) = \phi$ (i.e., independent split), then t_a and t_b conforms to definition 17 (atomic transactions).

Assertion 1: $Split_{t_a}[t_b]$ *is equivalent to* $Begin_{t_b}$ *if* $CanAccess(t_a, t_b) = DelegateSet(t_a, t_b) = \phi$.

This assertion points to an implementation that combines the semantics of Begin and Split events.

Lemma 4: Let t_a be the splitting transaction and t_b be the split transaction.
$((CanAccess(t_a, t_b) \neq \phi) \wedge (Commit_{t_b} \in H)) \Rightarrow (Commit_{t_a} \rightarrow Commit_{t_b})$.

That is, in the case of serial split, if both splitting and split transactions commit then the splitting transaction commits before the split transaction. This follows from axioms 13 and 14 in conjunction with Axioms 6-9.

Definition 23: A split transaction t_s initiated by the primary transaction t_p is a set of transactions t_i: $t_i = t_p \vee t_i \mathcal{AD}^* t_p$, where $\mathcal{AD}^*$ is induced by splits.

Theorem 2: A set of split transactions T is *serializable*.

To prove this we show (1) that the transactions constituting a split transaction t_s are serializable with each other using lemmas 3 and 4, (2) that they are serializable with respect to all other transactions using the semantics of delegation, lemma 2, and Axioms 15-17.

Definition 24: A split transaction management scheme is *correct* if it conforms to definition 22.

Sections 4.1 through 4.3 shown that ACTA can be used to prove the correctness of concurrency control algorithms for extended transactions similar to the serializability theory [3, 24] for traditional transactions.

5 Conclusions

As a conclusion, let us evaluate the ACTA formalism with respect to the motivating questions posed in the introduction. ACTA captures the (extended) functionality of a transaction model (1) by allowing the specification of significant events beyond commit and abort, (2) by allowing the specification of arbitrary transaction structures in terms of dependencies involving any significant event, (3) by supporting finer grain visibility for objects in the database by means of the Access and View Sets and the notion of delegation, (4) and by facilitating object-specific and transaction-specific semantic-based concurrency control.

In addition, the ACTA formalism facilitates comparisons between two transaction models at both the abstract level of their correctness properties, e.g. failure atomicity, serializability, objectwise serializability, setwise-serializability etc., and the realization level in terms of the significant events, structural properties etc.

Finally, whether two transaction models can be used in conjunction can be determined in ACTA by combining the characterizations of the two models and checking whether the new model retains the correctness properties of the two original ones. This was informally shown in [6] but it can easily be formalized using the formalism outlined here. This, as well as other examples of the use of the ACTA formalism, can be found in [8, 7].

In terms of future work, the primitives underlying ACTA suggest a set of primitive *mechanisms* that are required to provide adequate implementation support for building a flexible transaction system. Hence, it will be useful to utilize ACTA to identify the transaction management primitives required for a particular database application. In addition, ACTA can be used to show the correctness of a particular implementation by first formalizing the properties of the specific mechanisms used in the implementations and then showing that they will maintain the correctness properties of the model.

References

[1] Badrinath, B. and Ramamritham, K. Semantics-based concurrency control: Beyond Commutativity. In *Proceedings of the 4th IEEE Conference on Data Engineering*, pages 132-140, February 1987.

[2] Bancilhon, F., Kim, W., and Korth, H. A model of CAD Transactions. In *Proceedings of the 11th International Conference on VLDB*, pages 25-33, Stockholm, August 1985.

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

[4] Buchmann, A., Hornick, M., Markatos, E., and Chronaki, C. Specification of a Transaction Mechanism for a Distributed Active Object System. In *Proceedings of the OOPSLA/ECOOP 90 Workshop on Transactions and Objects*, pages 1-9, Ottawa, Canada, October 1990.

[5] Chrysanthis, P. K. and Ramamritham, K. A Unifying Framework for Transactions in Competitive and Cooperative Environments. *IEEE Bulletin on Office and Knowledge Engineering*, August 1990.

[6] 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.

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

[8] Chrysanthis, P. K. and Ramamritham, K. ACTA: The SAGA continues. In Elmagarmid, A. K., editor, *(tentative) Transaction Processing in Advanced Applications*. Morgan Kaufmann, 1991.

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

[10] Elmagarmid, A., Leu, Y., Litwin, W., and Rusinkiewicz, M. A Multidatabase Transaction Model for InterBase. In *Proceedings of the 16th International Conference on VLDB*, pages 507–518, August 1990.

[11] Elmagarmid A. (Issue Editor). Special Issue on Unconventional Transaction Management. *IEEE Technical Committee on Data Engineering*, 14(1), March 1991.

[12] Eswaran, K., Gray, J., Lorie, R., and Traiger, I. The Notion of Consistency and Predicate Locks in a Database System. *Communications of the ACM*, 19(11):624-633, November 1976.

[13] Fernandez, M. and Zdonik, S. Transaction Groups: A Model for Controlling Cooperative Transactions. In *Proceedings of the Workshop on Persistent Object Systems: Their Design, Implementation and Use*, pages 128–138, January 1989.

[14] Garcia-Molina, H., Gawlick, D., Klein, J., Kleissner, K., and Salem, K. Modeling Long-Running Activities as Nested Sagas. *IEEE Technical Committee on Data Engineering*, 14(1):14–18, March 1991.

[15] Garcia-Molina, H. and Salem, K. SAGAS. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 249–259, May 1987.

[16] Gray, J. The Transaction Concept: Virtues and Limitations. In *Proceedings of the 7th VLDB Conference*, pages 144–154, September 1981.

[17] Herlihy, M. P. and Weihl, W. Hybrid concurrency control for abstract data types. In *Proceedings of the 7th ACM symposium on Principles of Database Systems*, pages 201-210, March 1988.

[18] Kaiser, G. E. A Flexible Transaction Model for Software Engineering. In *Proceedings of the 6th International Conference on Data Engineering*, pages 560–567, Los Angeles, CA, February 1990.

[19] Korth, H. F., Levy, E., and Silberschatz, A. Compensating Transactions: A New Recovery Paradigm. In *Proceedings of the the 16th VLDB Conference*, pages 95–106, Brisbane, Australia, August 1990.

[20] Korth, H. F. and Speegle, G. Formal Models of Correctness without Serializability. In *Proceedings of the ACM SIGMOD International Conference on management of data*, pages 379–386, Chicago, Illinois, June 1988.

[21] Korth, H. F. and Speegle, G. Encapsulation of Transaction Management in Object Databases. In *Proceedings of the OOPSLA/ECOOP'90 Workshop on Transactions and Objects*, pages 27–32, Ottawa, Canada, October 1990.

[22] Martin, B. E. and Pedersen, C. Long-Lived Concurrent Activities. Technical Report HPL-90-178, HP Laboratories, october 1990.

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

[24] Papadimitriou, C. H. *The Theory of Database Concurrency Control.* Computer Science Press, 1986.

[25] Pu, C., Kaiser, G., and Hutchinson, N. Split-Transactions for Open-Ended activities. In *Proceedings of the 14th International Conference on VLDB*, pages 26–37, Los Angeles, California, September 1988.

[26] Schwarz, P. M. and Spector, A. Z. Synchronizing Shared Abstract Data Types. *ACM Transactions on Computer Systems*, 2(3):223–250, August 1984.

[27] Skarra, A. Localized Correctness Specifications for Cooperating Transactions in an Object-Oriented Database. *IEEE Bulletin on Office and Knowledge Engineering*, Summer 1990.

[28] Vinter, S., Ramamritham, K., and Stemple, D. Recoverable Actions in Gutenberg. In *Proceedings of the 6th International Conference on Distributed Computing Systems*, pages 242–249, May 1986.

[29] Weihl, W. *Specification and Implementation of Atomic Data Types.* PhD thesis, Massachusetts Institute of Technology, 545 Technology Square, Cambridge, MA, March 1984.