Synthesis of Extended Transaction Models using ACTA [1]

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

April 29, 1993

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 presents *ACTA as a tool for the synthesis of extended transaction models*, one which supports the development and analysis of new extended transaction models in a systematic manner. Here, this is demonstrated by deriving new transaction definitions (1) by modifying the specifications of existing transaction models, (2) by combining the specifications of existing models and (3) by starting from first principles. To exemplify the first, new models are synthesized from *atomic transactions* and *join transactions*. To illustrate the second, we synthesize a model that combines aspects of the *nested* and *split transaction* models. We demonstrate the latter by deriving the specification of an *open nested transaction* model from high-level requirements.

[1]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.

Contents

1 Introduction

Although powerful, the transaction model adopted 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*. To facilitate the formal description of transaction properties in an extended transaction model, we have developed ACTA[2], a comprehensive transaction framework. Specifically, using ACTA, one can specify and reason about the nature of interactions between extended transactions in a particular model. ACTA characterizes the semantics of interactions (1) in terms of different types of dependencies between transactions (e.g., commit dependency and abort dependency) and (2) in terms of transactions' effects on objects (their state and concurrency status, i.e., synchronization state). Through the former, one can specify relationships between significant (transaction management) events, such as *begin, commit, abort, delegate, split*, and *join*, pertaining to different transactions. Also, conditions under which such events can occur can be specified precisely. Transactions' effects on object's state and status are specified by associating a *view* and a *conflict set* with each transaction and by stating how these are affected when significant events occur. A view of a transaction specifies the state of objects visible to that transaction while the transaction's conflict set contains those operations with respect to which conflicts need be considered.

In [8, 6], we introduced the formalism underlying ACTA and demonstrated its expressive power by using it to define extended transaction models in an axiomatic form, specify correctness properties of the models, and prove that a particular model satisfies the specified properties. This paper presents *ACTA as a tool for the synthesis of extended transaction models*, one which supports the development and analysis of new extended transaction models in a systematic manner.

New transaction definitions can be derived either by tailoring existing transaction models, or by starting from first principles. As examples of the former we develop *Chain Transactions* (Section 4.1.2), *Reporting transactions* (Section 4.1.3) and *Co-Transactions* (Section 4.1.4) by modifying the specification of joint transactions [17], and derive the *nested-split* transaction model (Section 4.2) by combining the specifications of nested and split transaction models [16, 17]. As an example of the latter, we synthesize in Section 4.3 an *open nested* transaction model from the high-level requirements on transactions adhering to the model.

[2]We chose the name *ACTA*, meaning *actions* in Latin, given the framework's appropriateness for expressing the properties of actions used to compose a computation.

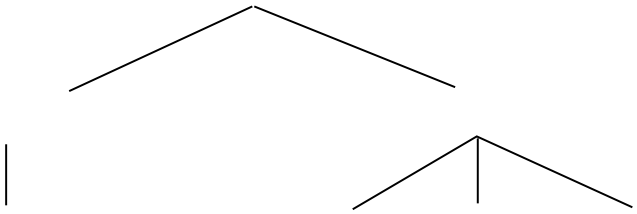

Figure 1: Dimensions of the ACTA Framework

2 The ACTA Formal Framework

ACTA is a first-order logic based formalism. It has five simple building blocks: History, dependencies between transactions, the *view* of a transaction, the *conflict set* of a transaction, and delegation.

This section provides a concise yet complete introduction to ACTA and its formal underpinnings. Section 2.1 provides some of the preliminary concepts underlying the ACTA formalism whereas Section 2.2 focuses on the concept of history which is central to the formalism. ACTA allows the specification of the effects of transactions on other transactions and also their effect on objects by means of constraints on histories. Inter-transaction dependencies, discussed in Section 2.3, forms the basis for the former while visibility of and conflicts between operations on objects, discussed in Section 2.4, form the basis for the latter. We will use examples from various extended transaction models to illustrate the concepts.

2.1 Preliminaries

2.1.1 Object Events

A database is the entity that contains all the shared objects in a system. A transaction accesses and manipulates the objects in the database by invoking operations specific to individual objects. 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 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 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.

DEFINITION 2.1: Invocation of an operation on an object is termed an *object event*. The type of an object defines the operations and thus, 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[3] by transaction t (i.e., $p_t[ob] \in OE_t$).

The effects of an operation on an object are not made permanent at the time of the execution of the operation. They need to be explicitly *committed* or *aborted*.

The effects of an operation p invoked by a transaction t on an object ob are made permanent in the database when $p_t[ob]$ is committed.

The effects of an operation p invoked by a transaction t on an object ob are obliterated when the $p_t[ob]$ is aborted.

Depending on the semantics of the operations and on the object's recovery properties, aborting an operation may force the abortion of other operations as well.

Commit and *Abort* operations are defined on every object for every operation. Invoked operations that have neither committed nor aborted are termed *in progress* operations. Typically, an operation is committed only if the invoking transaction commits and it is aborted only if the invoking transaction aborts. However, it is conceivable that an extended transaction may commit only a subset of its operations on an object while aborting the rest. Furthermore, through delegation (see Section 2.4), a transaction other than the *event-invoker*, i.e., the transaction that invoked an operation, can be granted the responsibility to commit or abort the operation.

2.1.2 Significant 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[4].

[3] We will use "invoke an event" to mean "cause an event to occur." One of the meanings of the word "invoke" is "to bring about."

[4] As shown in Section 2.4, in ACTA, the ability of a nested subtransaction to make its effect visible to its parent is specified by means of the notion of delegation.

Other transaction management primitives include Spawn, found in the nested transaction model, and Split and Join, found in the split transaction model [17].

DEFINITION 2.2: Invocation of a transaction management primitive is termed a *significant event.* A transaction model defines the significant events that can be invoked by transactions adhering to that model. SE_t denotes the set of significant events relevant to transaction t.

ACTA provides the means by which significant events and their semantics can be specified.

It is useful to distinguish, given the set of significant events associated with a transaction t, between events that are relevant to the initiation of t and those that are relevant to the termination of t.

DEFINITION 2.3: *Initiation events*, denoted by IE_t, is a set of significant events that can be invoked to initiate the execution of transaction t. $IE_t \subset SE_t$.

DEFINITION 2.4: *Termination events*, denoted by TE_t, is a set of significant events that can be invoked to terminate the execution of transaction t. $TE_t \subset SE_t$.

For example, in the split transaction model, Begin and Split are transaction initiation events whereas Commit, Abort and Join are transaction termination events.

A transaction is *in progress* if it has been initiated by some initiation event and it has not yet executed one of the termination events associated with it. A transaction *terminates* when it executes a termination event.

2.2 Histories and Conditions on Event Occurrences

Fundamental to ACTA is the notion of *history* [2] which represents the concurrent execution of a set of transactions T. ACTA captures the effects of transactions on other transactions and also their effects on objects through constraints on histories. Transaction models are defined in terms of a set of *axioms* which are invariant assertions about the histories generated by the transactions adhering to the particular model. Axioms can also be explicit *preconditions* or *postconditions* for operations and transaction management primitives. Consequently, the correctness properties of different transaction models can be expressed in terms of the properties of the histories produced by these models.

DEFINITION 2.5: A *history* H of the concurrent execution of a set of transactions T contains all the events, significant and object, invoked by the transactions in T and indicates the (partial) order in which these events occur.

DEFINITION 2.6: 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 $\epsilon' \in H$.)

H denotes the *complete* history. When a transaction invokes an event, that event is appended to the *current* history, denoted by H_{ct}. The partial order of the operations in a history pertaining to T is consistent with the partial order $\rightarrow$ of the events associated with each transaction t in T.

In general, we use ϵ_t to denote the invocation of an event ϵ, significant or object, by transaction t. We will omit the event-invoker when it is not important to specify the transaction which causes the event to occur in a history ($\epsilon \in H \Rightarrow \exists t\ \epsilon_t \in H$).

The occurrence of an event in a history can be affected 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 2.7: $(\epsilon \in H) \Rightarrow Condition_H$, where $\Rightarrow$ denotes *implication*, specifies that the event ϵ can belong to history H *only 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 2.8: $Condition_H \Rightarrow (\epsilon \in H)$ 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.

Consider $(\epsilon \rightarrow \epsilon') \Rightarrow (\alpha \in H)$. This states that *if* event ϵ occurs before ϵ' *then* event α belongs to the history.

2.3 Effects of Transactions on Other Transactions

Dependencies provide a convenient way to specify and reason about the behavior of concurrent transactions and can be precisely expressed in terms of the significant events associated with the transactions. Basically, dependencies are contraints on the histories produced by the concurrent execution of interdependent transactions. In the rest of this section, after formally specifying different types of dependencies, we identify the source of these dependencies.

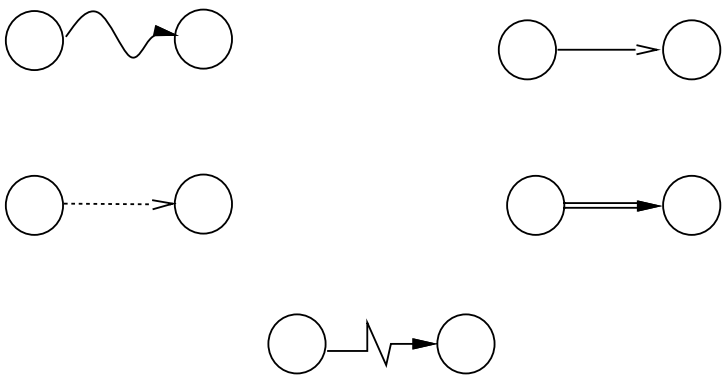

Figure 2: Inter-transaction Dependencies Graphs

2.3.1 Types of Dependencies

Let t_i and t_j be two extended transactions and H be a finite history which contains all the events pertaining to t_i and t_j.

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

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

Weak-Abort Dependency ($t_j \; \mathcal{WD} \; t_i$): if t_i aborts and t_j has not yet committed, then t_j aborts. In other words, if t_j commits and t_i aborts then the commitment of t_j precedes the abortion of t_i in a history; i.e.,
$\mathsf{Abort}_{t_i} \in H \Rightarrow (\neg(\mathsf{Commit}_{t_j} \rightarrow \mathsf{Abort}_{t_i}) \Rightarrow (\mathsf{Abort}_{t_j} \in H))$.

We would like to note that this list of dependencies involving the Commit and Abort events is *not* exhaustive. Other dependencies that involve significant events besides these events, can be defined. As new significant events are associated with extended transactions, new dependencies may be specified in a similar manner (e.g., see [6]). In this sense, ACTA is an open-ended framework.

Besides the logical representation introduced above, inter-transaction dependencies can be expressed in a pictorial form as graphs whose vertices represent transactions and arcs of different shapes represent different dependencies. We refer to such graphs as *dependency graphs*. Figure 2 shows the pictorial representation of the dependencies defined above and in Section 4.2. In general, dependency graphs can be more illustrative than the

corresponding sets of axioms in expressing the structure of extended transactions, such as the explicit nesting structure of nested transactions. (As discussed in the next section, one source of dependencies is the structure of extended transactions.) Through dependency graphs, it is possible to capture both the static structure as well as the dynamics of the evolution of the structure of transactions. The structure of transactions evolves as significant events inducing inter-transaction dependencies occur.

2.3.2 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 established at the time the child is *spawned*. This is expressed by a child transaction t_c establishing a weak-abort dependency on its parent t_p (t_c $\mathcal{WD}$ t_p) and a parent establishing a commit dependency on its child (t_p $\mathcal{CD}$ t_c).

$$\mathsf{Spawn}_{t_p}[t_c] \in H \Rightarrow (t_c \ \mathcal{WD} \ t_p) \wedge (t_p \ \mathcal{CD} \ t_c).$$

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 4.2.1 for a precise formal definition of nested transactions.) The commit dependency ensures that an orphan, i.e., a child transaction whose parent has terminated, will not commit.

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 [2] 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 operation 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.

2.4 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.

2.4.1 Conflicts between Operations and the Induced Dependencies

A history $H^{(ob)}$ of operation invocations on an object ob, $H^{(ob)} = p_1 \circ p_2 \circ ... \circ p_n$, indicates both the order of execution of the operations, (p_i precedes p_{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.

DEFINITION 2.9: Two operations p and q *conflict* in a state produced by $H^{(ob)}$, denoted by *conflict*$(H^{(ob)}, p, q)$, iff

$$\begin{aligned} (state(H^{(ob)} \circ p,\ q) &\neq state(H^{(ob)} \circ q,\ p)) \quad \vee \\ (return(H^{(ob)},\ q) &\neq return(H^{(ob)} \circ p,\ q)) \ \vee \\ (return(H^{(ob)},\ p) &\neq return(H^{(ob)} \circ q,\ p)). \end{aligned}$$

Two operations that do not conflict are *compatible*.

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, the semantics of the return values can be considered in dealing with conflicting operations.

DEFINITION 2.10: 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.

Interactions between conflicting operations can cause dependencies of different types between the invoking transactions. The type of interactions induced by conflicting operations depends on whether the effects of operations on objects are *immediate* or *deferred*. An operation has an immediate effect on an object only if it changes the state of the object as it executes and the new state is visible to subsequent operations. Thus, an operation p operates on the (most recent) state of the object, i.e., the state produced by the operation immediately preceding p. For example, effects are immediate in objects which perform *in-place updates* and employ logs for recovery. Effects of operations are *deferred* if operations are not allowed to change the state of an object as soon as they occur but, instead, the changes are effected only upon commitment of the operations. In this case, operations performed by a transaction are maintained in *intentions lists*. In the rest of the paper, we will consider the situation when the effects are immediate.

As mentioned earlier, in ACTA, the concurrency properties of an object are formally expressed in terms of *conflict relations* of the form:

$$conflict(p_{t_i}[ob], q_{t_j}[ob]) \wedge (p_{t_i}[ob] \rightarrow q_{t_j}[ob]) \Rightarrow Condition_H,$$

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. For instance, *commutativity* semantics of operations induce abort dependencies between conflicting operations:

$$conflict(p_{t_i}[ob], q_{t_j}[ob]) \wedge (p_{t_i}[ob] \rightarrow q_{t_j}[ob]) \Rightarrow (t_j \ \mathcal{AD} \ t_i).$$

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[5].

Commutativity does not distinguish between return-value dependent and return-value independent conflicts, but if it did then a weaker conflict notion, called *recoverability* [1], would result to a weaker conflict relation between return-value independent conflicting operations where $\mathcal{CD}$ is induced rather than $\mathcal{AD}$:

[5]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 a specific commit ordering. One way to avoid this is to force the invoking transaction to (a) wait until 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 in-progress operations and of dependencies that have been induced by the conflict. A commonly used synchronization mechanism for keeping track of in-progress operations and dependencies is based on (logical) *locks*.

$$\mathit{conflict}(p_{t_i}[ob], q_{t_j}[ob]) \wedge \mathit{return\text{-}value\text{-}independent}(p, q) \wedge (p_{t_i}[ob] \rightarrow q_{t_j}[ob]) \Rightarrow (t_j \; \mathcal{CD} \; t_i).$$

The generality of the conflict relations allows ACTA to capture different types of type-specific concurrency control discussed in the literature [18, 13, 21, 1, 7], and even to tailor them for cooperative environments [11, 19].

2.4.2 Controlling Object Visibility

View of a Transaction

As defined earlier, visibility refers to the ability of one transaction to see the effects of another transaction on objects *while* they are executing. ACTA allows finer control over the visibility of objects by associating two entities, namely, *view* and *conflict set*, with every transaction.

DEFINITION 2.11: The *view* of a transaction, denoted by $View_t$, specifies the objects and the *state* of objects visible to transaction t at a point in time.

This implies that that view specifies what objects can be operated on by a transaction. In addition, view specifies the state of these objects that is visible to the operations invoked by the transaction.

$View_t$ is formally a projection of a history where the projected events satisfy some $\mathcal{P}redicate$, typically involving H_{ct}:

$$View_t = Projection(H_{ct}, \mathcal{P}redicate).$$

In other words, $View_t$ is the subhistory constructed by eliminating any events in H_{ct} that do not satisfy the given $\mathcal{P}redicate$ while preserving the partial ordering of events in the view. For example, the view of a subtransaction t_c in the nested transaction model is defined to be the current history, i.e., $View_{t_c} = H_{ct}$. This states that (the effects of) all the events that have occurred thus far are visible to t_c meaning that t_c can view *the* most recent state of objects in the database.

For a slightly more elaborate example, suppose that a subtransaction t_c is restricted to view, at any given moment during its execution, only those objects that have been accessed by its parent t_p. The view of such a subtransaction t_c is defined as follows.

$$View_{t_c} = Projection(H_{ct}, (\forall t, ob, q \; q_t[ob] \mid \exists r \; r_{t_p}[ob] \in H_{ct})).$$

That is, the view of t_c is the history projected to contain all the operations q invoked by any transaction t on any object ob on which t_p has performed some operation r.

Conflict Set of a Transaction

A transaction t can invoke an operation on an object without conflicting with another operation invoked by transaction t_i if the operation performed by t_i is in the view of t but it is not included in the conflict set of t.

DEFINITION 2.12: The *conflict set* of a transaction t, denoted by $ConflictSet_t$, contains those operations in the current history with respect to which conflicts have to be determined when t invokes an operation.

The $ConflictSet_t$ is a subset of the object events in H_{ct} that satisfy some $\mathcal{P}redicate$:

$ConflictSet_t = \{p_{t_i}[ob] \mid \mathcal{P}redicate\}$.

For example, let us consider nested transactions once again. In nested transactions, a subtransaction t_c can access without conflicts any object currently accessed by one of its ancestors t_a. This is captured by:

$ConflictSet_{t_c} = \{p_{t_i}[ob] \mid \mathcal{I}nprogress(p_{t_i}[ob]) \wedge t_i \neq t_c \wedge t_i \notin Ancestor(t_c)\}$;
$Ancestor(t_c)$ is the set of ancestors of t_c.
$\mathcal{I}nprogress(p_{t_i}[ob])$ is *true* with respect to current history H_{ct} if $p_{t_i}[ob]$ has been performed but has neither committed nor aborted yet; i.e.,

$$\mathcal{I}nprogress(p_{t_i}[ob]) \Rightarrow ((p_{t_i}[ob] \in H_{ct}) \wedge ((Commit_{t_i}[p_{t_i}[ob]] \notin H_{ct}) \wedge (Abort_{t_i}[p_{t_i}[ob]] \notin H_{ct}))).$$

This states that an operation p invoked by t_i on an object ob is considered to conflict with an operation invoked by a child transaction t_c only if t_i and t_c are different, t_i is not an ancestor (in the nested transaction structure) of t_c, and p is still in progress. In other words, any operation invoked by an ancestor of t_c is not contained in $ConflictSet_{t_c}$. For this reason, a transaction t_c can invoke an operation that conflicts with another in progress, invoked by its ancestor t_a, without forming a dependency.

The axiomatic definition of a transaction model specifies the $View_t$ and $ConflictSet_t$ of each transaction t in that model. These determine if a new event can be invoked. Specifically, the preconditions of the event derived from the axiomatic definition of its invoking transaction are evaluated with respect to H_{ct} using the $View_t$ and $ConflictSet_t$. If its preconditions are satisfied, the new event is invoked and appended to the H_{ct} reflecting

its occurrence. The axiomatic definitions also specify how new dependencies may be established. As we saw earlier, if an event is an object event, the operation semantics may also induce new dependencies.

Delegation by a Transaction

The final building block of ACTA is *Delegation*. Traditionally, the invoker of an operation has the responsibility for committing or aborting the operation. In general, however, the operation invoker and the one committing the operation may be different.

DEFINITION 2.13: $ResponsibleTr(p_{t_i}[ob])$ identifies the transaction responsible for committing or aborting the operation $p_{t_i}[ob]$ with respect to the current history H_{ct}.

In general, a transaction may *delegate* some of its responsibilities to another transaction. More precisely,

DEFINITION 2.14: $Delegate_{t_i}[t_j, p_{t_i}[ob]]$ denotes that t_i delegates to t_j the responsibility for committing or aborting operation $p_{t_i}[ob]$.

More generally, $Delegate_{t_i}[t_j, DelegateSet]$ denotes that t_i delegates to t_j the responsibility for committing or aborting each operation in the $DelegateSet$.

Delegation has the following ramifications which are formally stated in [6]:

- $ResponsibleTr(p_{t_i}[ob])$ 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 $Abort_{t_j}[p_{t_i}[ob]]$ is that $ResponsibleTr(p_{t_i}[ob])$ is t_j. Similarly, a precondition for the event $Commit_{t_j}[p_{t_i}[ob]]$ is that $ResponsibleTr(p_{t_i}[ob])$ is t_j. Hence, from now on, unless essential, we will drop the subscript, e.g., t_j, associated with the operation abort and commit events.

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

- From the prespective of dependencies, once an operation is delegated, it is as though the delegatee performed the operation. Thus, delegation redirects the dependencies induced by delegated operations from the delegator to the delegatee — dependencies are sort of responsibilities.

Note that delegation 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.

In controlling visibility, we will find it useful to associate each transaction with an *access set.*

DEFINITION 2.15: $AccessSet_t = \{p_{t_i}[ob] \mid ResponsibleTr(p_{t_i}[ob]) = t\}$; i.e., $AccessSet_t$ contains all the operations for which t is responsible.

In nested transactions, when the root commits, its effects are made permanent in the database, whereas when a subtransaction commits, via inheritance, its effects are made visible to its parent transaction. 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 operations that it is responsible for

$$\mathsf{Commit}_{t_c} \in H \Leftrightarrow Delegate_{t_c}[t_p, AccessSet_{t_c}] \in H.$$

Delegation need not occur only upon commit or abort but a transaction can delegate any of the operations in its access set to another transaction at any point during its execution. This is the case for Co-Transactions and Reporting Transactions described in Section 4.

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.

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

3 A Simple Example of ACTA Specification: 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. Below we first define the correctness properties of objects starting with the serializability correctness criterion.

Let $\mathcal{C}$ be a binary relation on transactions in T.

Let H be the history of events relating to transactions in T.

DEFINITION 3.1: *Serializability*

$\forall t_i, t_j \in T, t_i \neq t_j$
$(t_i\ \mathcal{C}\ t_j)$ iff $\exists ob\ \exists p, q\ (\mathit{conflict}(p_{t_i}[ob], q_{t_j}[ob]) \wedge (p_{t_i}[ob] \rightarrow q_{t_j}[ob]))$

Let $\mathcal{C}^*$ be the transitive-closure of $\mathcal{C}$; i.e.,
$(t_i\ \mathcal{C}^*\ t_k)$ iff $[(t_i\ \mathcal{C}\ t_k) \vee \exists t_j\ (t_i\ \mathcal{C}\ t_j \wedge t_j\ \mathcal{C}^*\ t_k)]$.

H is *(conflict) serializable* iff $\forall t \in T\ \neg(t\ \mathcal{C}^*\ t)$

DEFINITION 3.2: *Objects' Correctness*

An object *ob* behaves *correctly* iff
$\forall t_i, t_j, t_i \neq t_j,\ \forall p, q$
$(\mathit{return\text{-}value\text{-}dependent}(\mathrm{p,q}) \wedge (p_{t_i}[ob] \rightarrow q_{t_j}[ob])) \Rightarrow$
$((\mathit{Abort}[p_{t_i}[ob]] \in H^{(ob)}) \Rightarrow (\mathit{Abort}[q_{t_j}[ob]] \in H^{(ob)}))$.

An object *ob* behaves *serializably* iff
$\forall t \in T\ \forall p\ (\mathit{Commit}[p_t[ob]] \in H^{(ob)}) \Rightarrow \neg(t\ \mathcal{C}^*\ t)$.

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

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. This ensures the correct behavior of objects in the presence of failures assuming immediate effects of operations on objects. Similarly, such dependencies can be defined for deferred effects.

A serializable behavior of an object is ensured by preventing committed transactions from forming cyclic $\mathcal{C}$ relationships.

DEFINITION 3.3: Transaction t is *failure atomic* if

1. $\exists ob\ \exists p\ (Commit[p_t[ob]] \in H) \Rightarrow$
   $\forall ob'\ \forall q\ ((q_t[ob'] \in H) \Rightarrow (Commit[q_t[ob']] \in H))$
2. $\exists ob\ \exists p\ (Abort[p_t[ob]] \in H) \Rightarrow$
   $\forall ob'\ \forall q\ ((q_t[ob'] \in H) \Rightarrow (Abort[q_t[ob']] \in H))$

As mentioned earlier, failure atomicity implies that all or none of a transaction's operations are committed (by some trasnaction). In the above definition, the "all" clause is captured by condition 1 which states that if an operation invoked by a transaction t is committed on an object, all the operations invoked by t are committed. The "none" clause is captured by condition 2 which states that if an operation invoked by a transaction t is aborted on an object, all the operations invoked by t are aborted. Note that failure atomicity does not require an operation to be committed or aborted by the invoking transaction.

In the same way that serializability and failure atomicity were expressed above, other correctness properties of extended transactions, such as, *quasi serializability* [10] and *predicatewise serializability* [14] can be expressed in ACTA [6].

Recall that each transaction model defines a set of significant events that transactions adhering to that model can invoke in addition to the invocation of operations on objects. A transaction is always associated with a set of initiation significant events that can be invoked to initiate the execution of the transaction, and a set of termination significant events that can be invoked to terminate the execution of the transaction. A set of *Fundamental Axioms* which is applicable to all transaction models specifies the relationship between significant events of the same or different type, and between significant events and operations on objects.

DEFINITION 3.4: FUNDAMENTAL AXIOMS OF TRANSACTIONS
Let t be a transaction and H^t the projection of the history H with respect to t.

I. $\forall \alpha \in IE_t\ (\alpha \in H^t) \Rightarrow \nexists \beta \in IE_t\ (\alpha \rightarrow \beta)$

II. $\forall \delta \in TE_t\ \exists \alpha \in IE_t\ (\delta \in H^t) \Rightarrow (\alpha \rightarrow \delta)$

III. $\forall \gamma \in TE_t\ (\gamma \in H^t) \Rightarrow \nexists \delta \in TE_t\ (\gamma \rightarrow \delta)$

IV. $\forall ob\ \forall p\ (p_t[ob] \in H) \Rightarrow ((\exists \alpha \in IE_t\ (\alpha \rightarrow p_t[ob])) \wedge (\exists \gamma \in TE_t\ (p_t[ob] \rightarrow \gamma)))$

Axiom I prevents a transaction from being initiated by two different events. Axiom II states that if a transaction has terminated, it must have been previously initiated. Axiom III prevents a transaction from being terminated by two different termination events. The

last axiom, Axiom IV, states that only in-progress transactions can invoke operations on objects.

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

DEFINITION 3.5: AXIOMATIC DEFINITION OF ATOMIC TRANSACTIONS
t denotes an atomic transaction.

1. $SE_t = \{\mathsf{Begin}, \mathsf{Commit}, \mathsf{Abort}\}$
2. $IE_t = \{\mathsf{Begin}\}$
3. $TE_t = \{\mathsf{Commit}, \mathsf{Abort}\}$
4. t satisfies the fundamental Axioms I to IV
5. $View_t = H_{ct}$
6. $ConflictSet_t = \{p_{t'}[ob] \mid t' \neq t, \mathcal{I}nprogress(p_{t'}[ob])\}$
7. $\forall ob\ \exists p\ p_t[ob] \in H \Rightarrow (ob\ is\ atomic)$
8. $\mathsf{Commit}_t \in H \Rightarrow \neg(t\ \mathcal{C}^*\ t)$.
9. $\exists ob\ \exists p\ Commit_t[p_t[ob]] \in H \Rightarrow \mathsf{Commit}_t \in H$
10. $\mathsf{Commit}_t \in H \Rightarrow \forall ob\ \forall p\ (p_t[ob] \in H \Rightarrow Commit_t[p_t[ob]] \in H)$
11. $\exists ob\ \exists p\ Abort_t[p_t[ob]] \in H \Rightarrow \mathsf{Abort}_t \in H$
12. $\mathsf{Abort}_t \in H \Rightarrow \forall ob\ \forall p\ (p_t[ob] \in H \Rightarrow Abort_t[p_t[ob]] \in H)$

Axiom 1 states that atomic transactions are associated with the three significant events: Begin, Commit and Abort. Axiom 2 specifies that Begin is the initiation event for atomic transactions. Axiom 3 indicates that Commit and Abort are the termination events associated with atomic transactions. Axiom 4 states that atomic transactions satisfy the fundamental axioms.

Axiom 5 specifies that a transaction sees the current state of the objects in the database. Axiom 6 states that conflicts have to be considered against all in-progress operations performed by different transactions. 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. Axiom 8 states that an atomic transaction can commit only if it is not part of a cycle of $\mathcal{C}$ relations developed through the invocation of conflicting operations. Note that the atomicity property local to individual objects is not sufficient to guarantee serializable execution of concurrent transactions across all objects [20]. Axiom 9 states that if an operation is committed on an object, the invoking transaction must commit, and Axiom 10 states that if a transaction commits, all the operations invoked by the transaction are committed.

Axioms 8, 9 and 10 define the semantics of the Commit event of atomic transactions in terms of the *Commit* operation defined on objects. Similarly, Axioms 11 and 12 define

the semantics of the Abort event in terms of the *Abort* operation defined on objects. Axiom 11 states that if an operation is aborted on an object, the invoking transaction must abort, and Axiom 12 states that if a transaction aborts, all the operations invoked by the transaction are aborted.

Based on the above axioms, the failure atomicity and serializability properties of atomic transactions can be shown (see [6]).

4 Synthezing New Transaction Models

Below we synthesize two new families of extended transaction models. The first is derived from the *joint transaction* model [17]. The second is derived from the *nested transaction* model [16] and the *split transaction* model [17]. We also synthesize a new open nested transaction model starting from first principles and high-level requirements.

A common characteristic of these new extended transaction models is that they support *delegation* between transactions. The following definition of conflicts takes into account the presence of delegation.

DEFINITION 4.1: Let $\mathcal{C}_N$ be a binary relation on transactions, and t_i and t_j be transactions.

$(t_i\ \mathcal{C}_N\ t_j), t_i \neq t_j$ iff

$\exists ob\ \exists p, q\ \exists t_m, t_n\ (conflict(p_{t_m}[ob], q_{t_n}[ob]) \wedge (p_{t_m}[ob] \rightarrow q_{t_n}[ob]) \wedge$
$(ResponsibleTr(p_{t_m}[ob]) = t_i) \wedge (ResponsibleTr(p_{t_n}[ob]) = t_j))$

This definition extends the definition of the $\mathcal{C}$ relation [Definition 3.1] to include the serialization orderings due to the delegated objects. (To see that $\mathcal{C}_N$ is a generalization of $\mathcal{C}$, consider the case in which delegation does not occur. In the absence of delegation, $t_m = t_i$ and $t_n = t_j$.) In this way, by substituting $\mathcal{C}_N$ for $\mathcal{C}$ in the definition of serializability [Definition 3.1], transactions are serialized with respect to operations for which they are responsible.

DEFINITION 4.2: H is *(conflict) serializable* iff

$\forall t \in T\ \neg(t\ \mathcal{C}_N^*\ t)$

There is no need to revisit the definition of failure atomicity in face of delegation. Failure atomicity does not require the invoking transaction of an operation to be the transaction to either commit or abort the operation. Thus, failure atomicity [Definition 3.3] allows the possibility for all the operations invoked by a transaction and not delegated to another transaction to be committed (aborted) by the invoking transaction and for all the delegated operations to be committed (aborted) by the delegatees. However, the

examination of a transaction's failure semantics only with respect to the objects that the transaction is responsible for leads to a definition of another failure property which is weaker than failure atomicity.

DEFINITION 4.3: Transaction t is *quasi failure atomic* if

1. $\exists ob\ \exists p\ \exists t_i\ Commit_t[p_{t_i}[ob]] \in H \Rightarrow$
   $\forall ob'\ \forall q\ \forall t_j\ (q_{t_j}[ob'] \in AccessSet_t \Rightarrow Commit_t[q_{t_j}[ob']] \in H)$
2. $\exists ob\ \exists p\ \exists\ Abort_t[p_{t_i}[ob]] \in H \Rightarrow$
   $\forall ob'\ \forall q\ \forall t_j\ (q_{t_j}[ob'] \in AccessSet_t \Rightarrow Abort_t[q_{t_j}[ob']] \in H)$

According to this definition, a transaction t is quasi failure atomic if either "all" or "none" of the operations for which the transaction t is responsible are committed. Recall the $AccessSet_t$ contains all the operations for which t is responsible. (To recap, a transaction is failure atomic if all the operations it *invokes* are committed or none at all; a transaction is quasi failure atomic if all operations that it is *responsible* for are committed or none at all.) Clearly, in the absence of delegation quasi failure atomicity is equivalent to failure atomicity.

4.1 Joint Transaction Model and its Variations

In this section, we derive three new extended transaction models, namely, *chain transactions, reporting transactions* and *co-transactions*, though a series of manipulations, beginning with the axiomatic definition of joint transactions [17]. In [5], we defined these models using *dependency production rules*, a formalism close to dependency graphs which capture the static structure and the dynamics of the evolution of the structure of transactions. Here we use axiomatic definitions to express the properties of these transaction models.

4.1.1 Joint Transactions

In the joint transactions model, Join is a termination event (in addition to the standard Commit and Abort events). That is, it is possible for a transaction, instead of committing or aborting, to join another transaction. The joining transaction delegates its objects to the *joint* transaction. Thus, the effects of the joining transaction are made persistent in the database only when the joint transaction commits. Otherwise they are discarded. Thus, if the joint transaction aborts, the joining transaction is effectively aborted. A joint transaction can itself join another transaction.

Here are the basic properties of joint transactions, expressed in ACTA.

DEFINITION 4.4: AXIOMATIC DEFINITION OF JOINT TRANSACTIONS
t_a denotes a joining transaction.
t_b denotes a joint transaction.
t denotes either a joining or a joint transaction.

1. $SE_t = \{\mathsf{Begin}, \mathsf{Join}, \mathsf{Commit}, \mathsf{Abort}\}$
2. $IE_t = \{\mathsf{Begin}\}$
3. $TE_t = \{\mathsf{Join}, \mathsf{Commit}, \mathsf{Abort}\}$
4. t satisfies the fundamental Axioms I to IV
5. $View_t = H_{ct}$
6. $ConflictSet_t = \{p_{t'}[ob] \mid t' \neq t, \mathcal{I}nprogress(p_{t'}[ob])\}$
7. $\forall ob\ \exists p\ p_t[ob] \in H \Rightarrow (ob\ is\ atomic)$
8. $\mathsf{Commit}_t \in H \Rightarrow \neg(t\ C_N^*\ t)$
9. $\exists ob\ \exists q\ \exists t_i\ Commit_t[q_{t_i}[ob]] \in H \Rightarrow \mathsf{Commit}_t \in H$
10. $\mathsf{Commit}_t \in H \Rightarrow \forall ob\ \forall q\ \forall t_i\ (q_{t_i}[ob] \in AccessSet_t \Rightarrow Commit_t[q_{t_i}[ob]] \in H)$
11. $\exists ob\ \exists q\ \exists t_i\ Abort_t[q_{t_i}[ob]] \in H \Rightarrow \mathsf{Abort}_t \in H$
12. $\mathsf{Abort}_t \in H \Rightarrow \forall ob\ \forall q\ \forall t_i\ (q_{t_i}[ob] \in AccessSet_t \Rightarrow Abort_t[q_{t_i}[ob]] \in H)$
13. $\mathsf{Join}_{t_a}[t_b] \in H \Leftrightarrow Delegate_{t_a}[t_b, AccessSet_{t_a}] \in H$

Axiom 1 states that transactions in the joint transaction model are associated with four significant events, namely, Begin, Join, Commit and Abort. The Begin, Commit and Abort events have the same semantics as the corresponding events of the atomic transactions [Axioms 4–12].

Axiom 13 specifies that when Join occurs, the joining transaction's access set is delegated to the joint transaction. In this regard, a joining transaction behaves similar to child transaction in the nested transaction model when the child transaction commits (see Section 4.2.1).

We now state some of the failure and ordering properties of joint transactions. Their proof can be found in [6].

LEMMA 4.1: A transaction t in the joint transaction model is *quasi failure atomic.*

LEMMA 4.2: A transaction t in the joint transaction model *behaves like* an atomic transaction if t commits or aborts, i.e., if it does not join any other transaction, and has not been joint by any other transaction.

In other words, a joint transaction that commits or aborts is *failure atomic* and executes in a *serializable* manner.

THEOREM 4.1: A joining transaction t_a may *not* be serializable with respect to the joint transaction t_b.

COROLLARY 1: A joining transaction t_a is *serializable* with respect to the joint transaction t_b iff $\mathsf{Join}_{t_a}[t_b] \in H \Rightarrow \neg((t_a\ \mathcal{C}_N^*\ t_b) \wedge (t_b\ \mathcal{C}_N^*\ t_a))$.

4.1.2 Chain Transactions

A special case of joint transactions is one that restricts the structure of joint transactions to a linear chain of transactions. We can call these transactions *Chain Transactions*[6]. A chain transaction is formed initially by a traditional transaction joining another traditional transaction and subsequently by the joint transaction joining another traditional transaction. This is achieved by introducing an axiom to restrict the invocation of the Join event such that only linear structures result [Axiom 14].

DEFINITION 4.5: AXIOMATIC DEFINITION OF CHAIN TRANSACTIONS
t_a denotes a joining transaction.
t_b denotes a joint transaction.
t denotes either a joining or a joint transaction.

1..13. Axiom 1..13 of Definition 4.4.
14. $\mathsf{Join}_{t_a}[t_b] \in H \Rightarrow \nexists t\ (\mathsf{Join}_t[t_b] \rightarrow \mathsf{Join}_{t_a}[t_b])$

All the lemmas and theorems expressing the correctness properties of joint transactions (Section 4.1.1) hold also for chain transactions.

Chain transactions can more appropriately capture a reliable computation consisting of a varying sequence of tasks each of which executes possibly at a different site of a computer network. That is, each task is structured as a transaction. The beginning of the first transaction initiates the computation. The computation expands dynamically when a transaction completes its execution by joining another transaction and hence extending the sequence of transactions. The commitment of any transaction in the sequence successfully completes the computation. The abort of any transaction terminates the computation and due to quasi failure atomicity its effects together with those of all previous transactions in the sequence are obliterated.

4.1.3 Reporting Transactions

A variation of the joint transaction model is the transaction model in which Join is not a termination event ($\mathsf{Join} \notin TE_t$). A joining transaction continues its execution and

[6]Chain transactions are of a more general form than IBM's Chain transactions.

periodically *reports* its results to the joint transaction by delegating more operations to the joint transaction. We call these transactions *Reporting Transactions*. Reporting transactions must invoke either Commit or Abort to complete their computation [Axiom 3].

Here is the formal definition of reporting transactions in ACTA. Other than the axioms for the Join event, the axioms for the other significant events are the same as in the joint transaction model.

DEFINITION 4.6: AXIOMATIC DEFINITION OF REPORTING TRANSACTIONS
t_a denotes a joining transaction.
t_b denotes a joint transaction.
t denotes either a joining or a joint transaction.

1. $SE_t = \{\mathsf{Begin}, \mathsf{Join}, \mathsf{Commit}, \mathsf{Abort}\}$
2. $IE_t = \{\mathsf{Begin}\}$
3. $TE_t = \{\mathsf{Commit}, \mathsf{Abort}\}$

4..12. Axiom 4..12. of Definition 4.4.

13. $\mathsf{Join}_{t_a}[t_b] \in H \Leftrightarrow Delegate_{t_a}[t_b, ReportSet_{t_a}] \in H$
14. $\mathsf{Join}_{t_a}[t_b] \in H \Rightarrow (t_a \; \mathcal{AD} \; t_b)$
15. $(\mathsf{Join}_{t_a}[t_b] \in H) \Rightarrow \nexists t, t \neq t_b \; (\mathsf{Join}_{t_a}[t] \rightarrow \mathsf{Join}_{t_a}[t_b])$
16. $\mathsf{Join}_{t_a}[t_b] \in H \Rightarrow \mathsf{Join}_{t_b}[t_a] \notin H$

$ReportSet_{t_a}$ contains the operations on the objects to be delegated [Axiom 13]. $ReportSet_{t_a} \subseteq AccessSet_{t_a}$. Thus, reporting transactions may delegate some and not necessarily all of their operations on objects at the time of a join.

The abort-dependency induced by Axiom 14 effectively maintains the termination semantics of joining transactions in the joint transaction model by guaranteeing the abortion of the joining transaction t_a if the joint transaction t_b aborts. This is because Axiom 15 prevents t_a from joining more than one transaction. Furthermore, Axiom 16 prevents t_b from joining back t_a.

Reporting transactions provide a more interesting control structure than joint transactions and can be useful in structuring data-driven computations. Reporting transactions can be restricted to a linear form in a manner similar to chain transactions in which case they can support pipeline-like computations, or allowed to form more complex control structures by permiting a reporting transaction to join more than one transaction in which case they can support, for example, star-like computations.

4.1.4 Co-Transactions

The characterization of reporting transactions allows t_a to continue its execution but prevents t_b from joining t_a [Axiom 15]. Suppose t_a is suspended when it joins t_b and also t_b is allowed to join t_a. The transaction t_a can be effectively suspended, if, at the time of the join, its view becomes empty. With an empty view, t_a can no longer access any object in the system. We call this *view curtailment*. t_a will be able to resume execution when t_b joins t_a. This is because, after the join, t_a's view will be restored while t_b's is curtailed. We call these transactions *co-transactions* because they behave like *co-routines* in which control is passed from one transaction to the other transaction at the time of the delegation and they resume execution where they were previously suspended. In the co-transaction model specified below, the view of the co-transaction that resumes execution is restored to H_{ct}.

Clearly, in the co-transaction model, the Join event is not a termination event (Join $\notin TE_t$) and co-transactions must invoke either commit or abort in order to complete their execution [Axiom 3].

Here is the formal definition of co-transactions in ACTA:

DEFINITION 4.7: AXIOMATIC DEFINITION OF CO-TRANSACTIONS
t_a denotes a joining transaction.
t_b denotes a joint transaction.
t denotes either a joining or a joint transaction.

1. $SE_t = \{\mathsf{Begin}, \mathsf{Join}, \mathsf{Commit}, \mathsf{Abort}\}$
2. $IE_t = \{\mathsf{Begin}\}$
3. $TE_t = \{\mathsf{Commit}, \mathsf{Abort}\}$

4..14. Axiom 4..14 of Definition 4.6

15 $post(\mathsf{Join}_{t_a}[t_b]) \Rightarrow (View_{t_a} = \phi) \wedge (View_{t_b} = H_{ct})$

16 $\mathsf{Join}_{t_a}[t_b] \in H \Rightarrow (t_b \; \mathcal{SCD} \; t_a)$

Here $\mathcal{SCD}$ stands for *strong commit* dependency whereby if t' commits, t'' must commit:
$(t'' \; \mathcal{SCD} \; t')$: $(Commit_{t'} \in H \Rightarrow Commit_{t''} \in H)$.

The termination semantics of co-transactions are captured by Axioms 14 and 16. According to the semantics of joint and reporting transactions, Axiom 14 ensures the abortion of the joining transaction t_a if the joint transaction t_b aborts. Axiom 16 states that if the joint transaction t_b commits, then the joining transaction t_a is also committed. Thus, both commit or neither.

Co-Transactions are useful in realizing applications that can be decomposed into interactive, and potentially distributed, subtasks which cannot execute in parallel. For

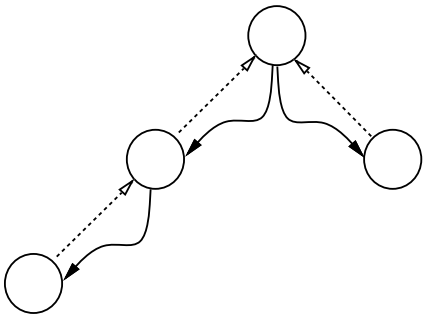

Figure 3: Structure of Nested Transactions

instance, co-transactions can be used in setting a meeting between two persons by having one co-transaction executing per person against the individual's calendar database. Co-transactions, as well as reporting transactions, can be easily modified to form more complex control structures in order to produce more interesting styles of cooperation.

4.2 Nested-Split Transaction Model

We first give the axiomatic definition of nested transactions and split transactions and then show how a combined model can be produced.

4.2.1 Nested Transactions

In the Nested Transaction model, e.g. [16], 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 and other non-related transactions and are failure atomic with respect to their parent. They 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 the database is also potentially accessible to the subtransaction. When a subtransaction commits, the objects modified by it are made accessible to its parent transaction and the effects on the objects are made permanent in a database only when the root transaction commits.

Now, let us define nested transactions using the ACTA formalism. $Ancestors(t)$ is the set of all ancestors of a transaction t whereas $Descendants(t)$ is the set of all descendants of t. $Parent(t)$ contains the parent transaction of t.

DEFINITION 4.8: AXIOMATIC DEFINITION OF NESTED TRANSACTIONS
t_0 denotes the root transaction. $Parent(t_0) = Ancestor(t_0) = \phi$
t_c denotes a subtransaction of t_p. $Parent(t_c) = t_p$
t_p denotes a root or a subtransaction.

1. SE_{t_0} = {Begin, Spawn, Commit, Abort}
2. IE_{t_0} = {Begin}
3. TE_{t_0} = {Commit, Abort}
4. SE_{t_c} = {Spawn, Commit, Abort}
5. IE_{t_c} = {Spawn}
6. TE_{t_c} = {Commit, Abort}
7. t_p satisfies the fundamental Axioms I to IV
8. $View_{t_p} = H_{ct}$
9. $ConflictSet_{t_0} = \{p_t[ob] \mid t \neq t_0, \mathcal{I}nprogress(p_t[ob])\}$
10. $\forall ob\ \exists p\ p_{t_p}[ob] \in H \Rightarrow (ob\ is\ atomic)$
11. $\mathsf{Commit}_{t_p} \in H \Rightarrow \neg(t_p\ C_N^*\ t_p)$
12. $\exists ob\ \exists p\ \exists t\ Commit_{t_p}[p_t[ob]] \in H \Rightarrow \mathsf{Commit}_{t_p} \in H \wedge Parent(t_p) = \phi$
13. $\mathsf{Commit}_{t_p} \in H \wedge Parent(t_p) = \phi \Rightarrow$
    $\forall ob\ \forall p\ \forall t\ (p_t[ob] \in AccessSet_{t_p} \Rightarrow Commit_{t_p}[p_t[ob]] \in H)$
14. $\exists ob\ \exists p\ \exists t\ Abort_{t_p}[p_t[ob]] \in H \Rightarrow \mathsf{Abort}_{t_p} \in H$
15. $\mathsf{Abort}_{t_p} \in H \Rightarrow \forall ob\ \forall p\ \forall t\ (p_t[ob] \in AccessSet_{t_p} \Rightarrow Abort_{t_p}[p_t[ob]] \in H)$
16. $Begin_{t_p} \in H \Rightarrow Parent(t_p) = \phi \wedge Ancestor(t_p) = \phi$
17. $ConflictSet_{t_c} = \{p_t[ob] \mid t \neq t_c, t \notin Ancestors(t_c), \mathcal{I}nprogress(p_t[ob])\}$
18. $\mathsf{Spawn}_{t_p}[t_c] \in H \Rightarrow Parent(t_c) = t_p$
19. $\mathsf{Spawn}_{t_p}[t_c] \in H \Rightarrow (t_c\ \mathcal{WD}\ t_p) \wedge (t_p\ \mathcal{CD}\ t_c)$
20. $\mathsf{Commit}_{t_c} \in H \Leftrightarrow Delegate_{t_c}[Parent(t_c), AccessSet_{t_c}] \in H$
21. $\forall t \in Descendants(t_p)\ \forall ob\ \forall p, q\ (p_t[ob] \rightarrow q_{t_p}[ob]) \Rightarrow$
    $\exists t_c\ ((Delegate_{t_c}[t_p, AccessSet_{t_c}] \rightarrow q_{t_p}[ob]) \wedge p_t[ob] \in AccessSet_{t_c})$
22. $Ancestor(t_c) = Ancestor(t_p) \cup \{t_p\} \wedge \forall t\ t_p \in Descedant(t) \Rightarrow t_c \in Descendant(t)$

The nested transaction model supports two types of transactions, namely, *root transactions* and *nested subtransactions*, which are associated with different significant events [Axioms 1 and 4]. The semantics of root transactions are similar to atomic transactions [Axioms 7–15]. The Abort event has the same semantics for both transaction types which are similar to those of the Abort in atomic transactions [Axioms 14 and 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 11–13]. In contrast, when a subtransaction commits, through *delegation*, the

operations in its access set are made persistent and visible only to its parent transaction [Axiom 20]. Axiom 20 which together with Axiom 11 define the semantics of the Commit event of subtransactions, clearly specifies that the commitment of a subtransaction does not imply the commitment of its operations and the operations that it is responsible for.

Spawn is used to initiate a new subtransaction. The Spawn event establishes a parent/child relationship between the spawning and spawned transactions [Axiom 18]. This relationship is reflected by the weak-abort dependency $\mathcal{WD}$ and commit dependency $\mathcal{CD}$ between the related transactions [Axiom 19]. The ability of a subtransaction to invoke operations without conflicting with the operations of its ancestor transactions is expressed by excluding all the operations performed by its ancestors from the conflict set of the subtransaction [Axiom 17].

Axiom 21 states that given transaction t and its ancestor t_p and operations p and q, t_p can invoke q after t invokes p if t_p is responsible for committing or aborting p. In other words, t_p cannot invoke q before p is delegated to t_p. In the absence of this restriction, it would be possible for t_p to develop an abort dependency on t (t_p $\mathcal{AD}$ t) by invoking an operation that conflicts with 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.

Based on the above axiomatic definition of nested transactions, the failure semantics and the serializability property of nested transactions can be shown [6]. For example, although Axioms 7, 10, and 11 are sufficient to ensure the serializability of atomic transactions, they are not in the case of nested transactions because of Axiom 17 which allows dependencies between a parent transaction and its children to be ignored. Thus, a parent and a child transaction are not serializable.

4.2.2 Split Transactions

In the Split Transaction model [17], a transaction t_a can split into transactions t_a and t_b. At the time of the split, operations invoked by t_a up to the split can be divided between t_a and t_b making each responsible for committing and aborting those operations assigned to them. In order to facilitate further data sharing between t_a and t_b, operations which remain the responsibility of t_a may be designated as not conflicting with operations invoked by t_b after the split, and hence, t_b can view the effects of these operations. Depending on whether or not such operations have been designated, a split may be *serial*, or may be

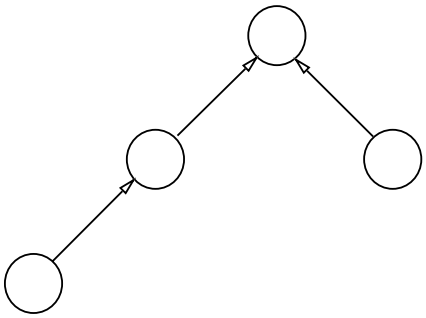

Figure 4: Structure of Split Transactions

independent. In the former case, t_a must commit in order for t_b to commit, whereas in the latter, t_a and t_b can commit or abort independently.

After the split, t_a can split again creating another split transaction t_c. Split transactions can further split creating new split transactions. A sequence of serial splits leads to a different type of hierarchically structured transactions from those of nested transactions. See Figure 4.

DEFINITION 4.9: AXIOMATIC DEFINITION OF SPLIT TRANSACTIONS
t_r denotes a primary transaction.
t_a denotes a splitting transaction, primary or split.
t_b denotes the split transaction of t_a.
t denotes a transaction, primary or split.

1. $SE_{t_r} = \{\mathsf{Begin}, \mathsf{Split}, \mathsf{Commit}, \mathsf{Abort}\}$
2. $IE_{t_r} = \{\mathsf{Begin}\}$
3. $TE_{t_r} = \{\mathsf{Commit}, \mathsf{Abort}\}$
4. $SE_{t_b} = \{\mathsf{Split}, \mathsf{Commit}, \mathsf{Abort}\}$
5. $IE_{t_b} = \{\mathsf{Split}\}$
6. $TE_{t_b} = \{\mathsf{Commit}, \mathsf{Abort}\}$
7. t satisfies the fundamental Axioms I to IV
8. $View_t = H_{ct}$
9. $ConflictSet_{t_r} = \{p_t[ob] \mid t \neq t_r, \mathcal{I}nprogress(p_t[ob])\}$
10. $\forall ob\ \exists p\ p_t[ob] \in H \Rightarrow (ob\ is\ atomic)$
11. $\mathsf{Commit}_t \in H \Rightarrow \neg(t\ C_N^*\ t)$
12. $\exists ob\ \exists q\ \exists t_i\ Commit_t[q_{t_i}[ob]] \in H \Rightarrow \mathsf{Commit}_t \in H$
13. $\mathsf{Commit}_t \in H \Rightarrow \forall ob\ \forall q\ \forall t_i\ (q_{t_i}[ob] \in AccessSet_t \Rightarrow Commit_t[q_{t_i}[ob]] \in H)$
14. $\exists ob\ \exists q\ \exists t_i\ Abort_t[q_{t_i}[ob]] \in H \Rightarrow \mathsf{Abort}_t \in H$

15. $\mathsf{Abort}_t \in H \Rightarrow \forall ob\ \forall q\ \forall t_i\ (q_{t_i}[ob] \in AccessSet_t \Rightarrow Abort_t[q_{t_i}[ob]] \in H)$
16. $\mathsf{Split}_{t_a}[t_b, CanAccess_{t_b}(t_a)] \in H \Rightarrow (CanAccess_{t_b}(t_a) \neq \phi \Rightarrow (t_b\ \mathcal{AD}\ t_a))$
17. $\mathsf{Split}_{t_a}[t_b, CanAccess_{t_b}(t_a)] \in H \Leftrightarrow Delegate_{t_a}[t_b, DelegateSet] \in H$
18. $\forall ob\ \exists p\ \exists t\ p_t[ob] \in DelegateSet \Rightarrow (\forall t'\ \forall q\ (ResponsibleTr(q_{t'}[ob]) = t_a \wedge (q_{t'}[ob] \rightarrow Delegate_{t_a}[t_b, DelegateSet])) \Rightarrow q_{t'}[ob] \in DelegateSet)$
19. $ConflictSet_{t_b} = \{p_t[ob] \mid (t \neq t_b, t \neq t_a, \mathcal{I}nprogress(p_t[ob])) \vee (t = t_a, \mathcal{I}nprogress(p_t[ob]) \wedge (p_t[ob] \notin CanAccess_{t_b}(t_a)))\}$
20. $\forall ob\ \forall p, q\ (\exists r\ (r_{t_a}[ob] \in CanAccess_{t_b}(t_a))) \wedge p_{t_a}[ob] \in H \Rightarrow (p_{t_a}[ob] \rightarrow q_{t_b}[ob])$

In the split transaction model, a transaction can be initiated through either the Begin event, called *primary* transaction, or the Split event, called *split* transaction. Although primary and split transactions are associated with different significant events [Axioms 1 and 4], their corresponding events share the same semantics [Axioms 11–15].

$\mathsf{Split}_{t_a}[t_b, CanAccess_{t_b}(t_a)]$ splits a primary or a split transaction t_a into a *splitting* transaction t_a and *split* transaction t_b. Since the idea is to allow the splitting transaction to give the split transaction the responsibility for finalizing some of its operations (these are the operations in the $DelegateSet$), the Split event is partially specified in terms of the delegation event $Delegate_{t_a}[t_b, DelegateSet]$ [Axiom 17]. To be more precise, a splitting transaction transfers to a split transaction the responsibility for all the operations on a particular object [Axiom 18]. That is, when a splitting transaction delegates an operation on an object *ob*, it delegates all the operations on *ob* that the splitting transaction is responsible for at the time of the split. Here, it is interesting to note that, in contrast to transactions initiated by the Begin event, through delegation, split transactions can affect objects in the database by committing or aborting delegated operations and without invoking any operation on them.

Further, the splitting transaction has the ability to allow the split transaction to view some of its operations on some objects without conflict (these are the operations in the $CanAccess_{t_b}(t_a)$) [Axiom 19]. However, the splitting transaction cannot view the operations of the split transaction on the same objects. A splitting transaction can continue to invoke operations on such objects as long as the split transaction has not invoked an operation on them [Axiom 20].

A split is *independent*, if $CanAccess_{t_b}(t_a)$ is empty. In the case of *serial split* in which $CanAccess_{t_b}(t_a)$ is not empty, t_b develops an abort dependency on t_a[7] [Axiom 16].

As in the case of nested transactions, Axioms 7, 10 and 11 are not sufficient to ensure serializability of split transactions due to Axioms 17 and 19. However, split transactions

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

are serializable as shown in [6]. That is, if t_a splits t_b serially, then t_a preceeds t_b in any serializable history in which both commit. If the spit is independent then t_a and t_b are serializable in any order. It should be pointed out that the above axiomatic definition of split transactions is more general than their original description which was within the context of lock-based concurrency control protocols.

4.2.3 Nested-Split Transactions

Given our definitions for atomic transactions (see Definition 3.5), nested transactions (see Definition 4.8) and split transactions (see Definition 4.9) in axiomatic form, it is not difficult to see which axioms reflect the differences between these models and which axioms capture their similarities.

For instance, the Begin, Abort, and Commit events in the split transaction model have the same semantics as those for the *root transactions* in the nested transaction model (which are the same as those of atomic transactions). However, although at first glance the Spawn event in nested transactions and the Split event in split transactions appear to have similar semantics, their precise definitions show the actual differences, for instance, in the induced dependencies. Specifically, whereas the Spawn event induces a commit dependency and a weak-abort dependency between the spawning and the spawned transactions [Axiom 18], the Split event induces an abort dependency of the split transaction on the splitting transaction [Axiom 19]. In addition, in contrast to the Spawn event, due to delegation, the Split event may associate a non-empty access set with the split transaction. Turning to similarities, it is possible to prove that both nested and split transactions produce only hierarchical transaction structures.

Given the similarities and differences between two models, the question of whether the two transaction models can be used in conjunction becomes important. Let us consider combining aspects from the nested and split transaction models. We would like to check whether the resulting model retains the properties of the two original models. This combination is derived by combining, where possible, nested transaction structures with split transaction structures, i.e., by considering how to handle existing dependencies, the view and the conflict set of the individual transactions.

Split-and-Nested Transactions

The obvious first approach is to merge the definitions of the two models. The resulting model is called *Split-and-nested Transaction* Model. In this model, given a nested transaction, it is possible to split the root or a subtransaction. A split transactions may further split creating another split transaction, or spawn a new subtransaction becom-

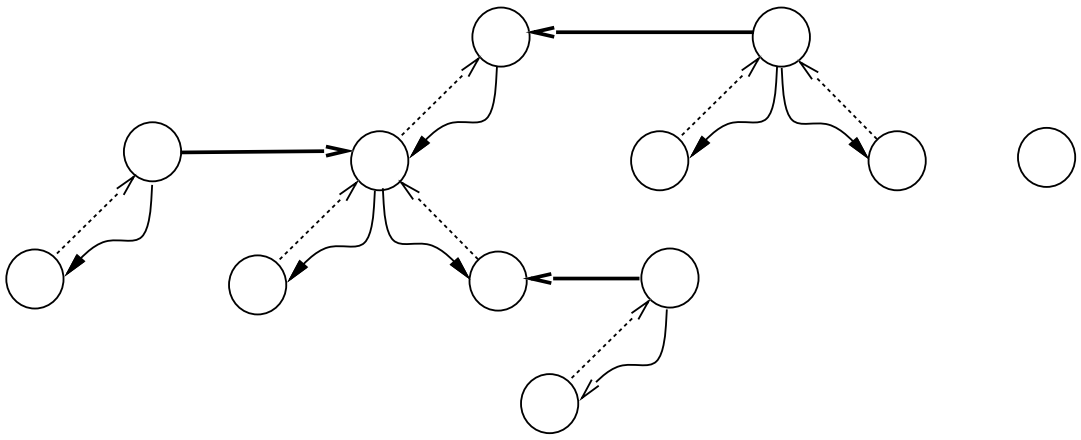

Figure 5: Structure of Split-and-Nested Transactions

ing a root of a new nested transaction. In this way, a set of possibly depended nested transactions may be created (See Figure 5).

DEFINITION 4.10: AN AXIOMATIC DEFINITION OF SPLIT-AND-NESTED TRANSACTIONS

t_0 denotes a root or a primary transaction. $Parent(t_0) = Ancestor(t_p) = \phi$.
t_c denotes a subtransaction of t_p. $Parent(t_c) = t_p$.
t_b denotes the split transaction of t_a. $Parent(t_b) = Ancestor(t_b) = \phi$.
t_p or t_a denotes a splitting transaction, root/primary, a subtransaction, or split.

1. SE_{t_0} = {Begin, Spawn, Split, Commit, Abort}
2. IE_{t_0} = {Begin}
3. TE_{t_0} = {Commit, Abort}
4. SE_{t_c} = {Spawn, Split, Commit, Abort}
5. IE_{t_c} = {Spawn}
6. TE_{t_c} = {Commit, Abort}
7. SE_{t_b} = {Spawn, Split, Commit, Abort}
8. IE_{t_b} = {Split}
9. TE_{t_b} = {Commit, Abort}

10..25. Axiom 7..22 of Definition 4.8.

26..30. Axiom 16..20 of Definition 4.9.

31. $\mathsf{Split}_{t_a}[t_b, CanAccess_{t_b}(t_a)] \in H \Rightarrow Parent(t_b) = \phi$

Axiom 31 ties together Split, a significant event that creates a new transaction not supported by nested transactions, with the notion of parent and ancestral transactions, not

present in split transactions in a way similar as in the case of Begin and Spawn events [Axioms 18 and 20 (or Axioms 16 and 18 of nested transactions)].

The split-and-nested transaction model produces only hierarchical transaction structures as the two original models. It involves the same dependencies between the various transaction types which are found in the original models. The additional abort dependency induced between a root or a subtransaction and its split transaction, in the case of serial split [Axiom 26 (or Axiom 16 of Split transactions)], does not violate the structure of nested transactions. Such abort dependencies between (sub)transactions of a nested transaction and other (sub)transactions are possible in the nested transaction model and may develop when transactions invoke conflicting operations on shared atomic objects [Axiom 10 of nested transactions].

Although this new model retains the properties of split transactions, it does not retain those of nested transactions. Specifically, split-and-nested transactions do not have the same ordering and failure properties of nested transactions. For instance, the split-and-nested transaction model allows the effects of subtransactions to be made permanent in the database by a transaction other than their ancestral root transaction.[8] To illustrate this, suppose a subtransaction t splits delegating to its split transaction t' an operation $p_t[ob]$. The delegated $p_t[ob]$ may be committed by t' since, when a split transaction commits, it commits all the operation in its AccessSet to the database [Axioms 16 and 31]. Furthermore, in the case of an independent split, it is possible for t (or its ancestral root transaction) to abort while $p_t[ob]$ is committed by t' and vice versa.

The split-and-nested transaction model is an example of an *open-nested* transaction model in which some component transactions (subtransactions) may decide to commit their effects in the database unilaterally. In Section 4.3, we will synthesize an open nested transaction model by precisely stating the requirements on the transactions adhering to the model.

Nested-Split Transactions

The split-and-nested transaction model defined above fails to retain the properties of nested transactions because the split-and-nested transaction model does not distinguish between splitting a root and a subtransaction. In the split-and-nested transaction model, it is possible for a subtransaction to split a root transaction. In fact, a split transaction is always a root transaction. However, the semantics of subtransactions are different from

[8] It can be proved (1) that operations invoked by subtransactions of a nested transaction are committed to the database only by the root transaction, and none of the subtransactions commit any operations, and (2) that if a root transaction aborts, all operations performed by the root and its descedants abort [6].

those of root transactions. This suggests that the semantics of a split transaction should be similar to those of its splitting transaction. Thus, when a root transaction splits, it should split into two root transactions, and when a subtransaction splits, it should split into two sibling subtransactions. In this way, a split of a subtransaction can no longer make any operations' effects permanent in the database but, as with any other subtransaction, when it commits, it delegates all operations in its access set to its parent transaction. We call such a derived model *Nested-Split Transaction* model. Nested-split transactions still retain the properties of split transactions in the sense that both a splitting and its split transaction exhibit the same behavior (i.e., their associated significant events have the same semantics) [Axioms 11 to 15 of split transactions].

The axiomatic definition of nested-split transactions can be derived from the definition of split-and-nested transactions by modifying Axioms 30 and 31, and by adding two new axioms, Axioms 32 and 33, one of which specifies the dependencies that are assumed to hold after a subtransaction is split into two subtransactions.

DEFINITION 4.11: AN AXIOMATIC DEFINITION OF NESTED-SPLIT TRANSACTIONS
t_0 denotes a root or a primary transaction. $Parent(t_0) = Ancestor(t_p) = \phi$.
t_c denotes a subtransaction of t_p. $Parent(t_c) = t_p$.
t_b denotes the split transaction of t_a. $Parent(t_b) = Ancestor(t_b) = \phi$.
t_p or t_a denotes a splitting transaction, root/primary, a subtransaction, or split.

1. $SE_{t_0} = \{$Begin, Spawn, Split, Commit, Abort$\}$
2. $IE_{t_0} = \{$Begin$\}$
3. $TE_{t_0} = \{$Commit, Abort$\}$
4. $SE_{t_c} = \{$Spawn, Split, Commit, Abort$\}$
5. $IE_{t_c} = \{$Spawn$\}$
6. $TE_{t_c} = \{$Commit, Abort$\}$
7. $SE_{t_b} = \{$Spawn, Split, Commit, Abort$\}$
8. $IE_{t_b} = \{$Split$\}$
9. $TE_{t_b} = \{$Commit, Abort$\}$

10..25. Axiom 7..22 of Definition 4.8.

26..29. Axiom 16..19 of Definition 4.9.

30. $\forall t, t = t_a \vee t \in Descendant(t_a)\ \forall ob\ \forall p, q$
$(\exists r\ (r_{t_a}[ob] \in CanAccess_{t_b}(t_a))) \wedge p_{t_a}[ob] \in H \Rightarrow (p_t[ob] \rightarrow q_{t_b}[ob])$
31. $\mathsf{Split}_{t_a}[t_b, CanAccess_{t_b}(t_a)] \in H \Rightarrow Parent(t_b) = Parent(t_a)$
32. $\mathsf{Split}_{t_a}[t_b, CanAccess_{t_b}(t_a)] \in H \Rightarrow$
$(Parent(t_a) \neq \phi \Rightarrow (t_b\ \mathcal{WD}\ Parent(t_a)) \wedge (Parent(t_a)\ \mathcal{CD}\ t_b))$

33. $\forall ob\ \exists t\ \exists p\ p_t[ob] \in DelegateSet \Rightarrow$
$(\forall t' \in Descendant(t_a)\ \forall q\ (q_{t'}[ob] \rightarrow Split_{t_a}[t_b, CanAccess_{t_b}(t_a)]) \Rightarrow$
$\exists t_c\ ((Delegate_{t_c}[t_a, AccessSet_{t_c}] \rightarrow Split_{t_a}[t_b, CanAccess_{t_b}(t_a)]) \wedge$
$q_{t'}[ob] \in AccessSet_{t_c})$

Axiom 30 corresponds to Axiom 20 of split transactions extended to take into account the descendants of a splitting transaction t_a which have the ability of invoking operations without conflicting with the operations of t_a. That is, the descendants of a splitting transaction as well as the splitting transaction itself, can continue to invoke operations on objects in the $CanAccess_{t_b}(t_a)$ as long as the split transaction has not invoked an operation on them.

Axiom 31 establishes the parent relationship of the split subtransaction by specifying that its parent is the parent of the subtransaction whose split it is.

Axiom 32 states that when a subtransaction t_a splits a transaction t_b, the dependencies between subtransaction t_a and its parent, say transaction t_p, are assumed to hold between t_b and t_p.

Axiom 33 states that in order for an operation on an object *ob* to be delegated at the time of a split, the splitting transaction should be responsible for all the operations on *ob* invoked by any of its descendant transactions. Consequently the split subtransaction is never delegated operations on objects which have been accessed by an active descendant of the splitting transaction. In the opposite case, the model would have required that the split subtransaction be considered an anscestor of the descendants of the splitting transactions due to Axiom 19.

Note that not all of the existing dependencies of splitting transaction are retained by the split transaction. For example, when a non-leaf subtransaction t_a splits, the dependencies between subtransaction t_a and its children are not assumed to hold between its split transaction t_b and t'_as children. The reason is that by establishing these dependencies either the hierarchical structure of the nested transactions is destroyed or some of the dependencies required by the nested transactions are eliminated. To illustrate this, consider the case of the independent split of a non-leaf subtransaction t_c into t_{c1} and t_{c2}. If the above dependencies were retained, a subtransaction t_d of t_c would have weak-abort dependencies on two ancestors, t_{c1} and t_{c2}, which is clearly disallowed by the hierarchical structure of the nested transaction model. The effects of retaining these dependencies are analyzed in [4].

Axioms 30 to 33 establish a sibling relationship between the splitting and split subtransactions. Hence, given a nested transaction, it is possible to split a root or any subtransaction while properties of both nested and split transactions are retained. Furthermore, due to delegation and the specification of CanAccess set at the time of a split,

two sibling transactions can effectively cooperate while they are still executing. In nested transactions, two sibling subtransactions cannot achieve cooperation while both siblings are active due to the conflict set specification of nested transactions (i.e., conflicts relative to the operations invoked by a transaction are not considered only by the descendants of the transaction). A nested subtransaction t can observe the effects of one of its siblings t' on an object without conflicts after t' has committed and delegated all its operations to their parent. Thus, nested-split transactions support a higher level of visibility between subtransactions than nested transactions making them a useful new transaction model for a cooperative environment. (A similar type of interaction occurs in the extended nested transaction model proposed in [15].)

4.3 Open Nested Transaction Model

In an open nested transaction model, component transactions may decide to commit or abort unilaterally. Assume that we need an open nested transaction model that supports two-level transactions with special components. Let s be a two-level transaction that has n component transactions, $t_1, .., t_n$. Some of the components are compensatable; each such t_i has a compensating transaction $comp\_t_i$ that semantically undoes the effects of t_i.

In order to derive the specification of this new transaction model, during synthesis we need to identify the different types of transactions which the model will support, the significant events associated with each type and the relationships among transactions. We will express these transaction relationships in terms of the significant events of the involved transactions. Also, for each type we need to define the visibility (i.e., view set) and conflict set of the transactions of the type and the semantics of the events associated with a particular transaction.

4.3.1 Specifying the Building Blocks

Let us begin the specification of this model by associating all transactions, components or otherwise, with the significant events {Begin, Commit, Abort}. Component and compensating transactions are atomic transactions with structure-induced inter-transaction dependencies.

Component transactions can commit without waiting for any other component or s to commit. However, if s aborts, a component transaction that has not yet committed will be aborted. We can capture this requirement using a weak-abort dependency:

$\forall 0 \leq i \leq n \ (t_i \ \mathcal{WD} \ s)$

Suppose some of the components of s are considered vital in that s is allowed to commit only if its *vital* components commit. These components are members of the set

VitalTrs. We can capture this requirement as follows:

$\forall 0 \leq i \leq n\ (t_i \in VitalTrs \Rightarrow (s\ \mathcal{AD}\ t_i))$.

If a vital transaction aborts, s will be aborted. Transaction s can commit even if one of its non-vital components aborts but s has to wait for them to commit or abort. This is expressed using a commit dependency.

$\forall 0 \leq i \leq n\ (t_i \notin VitalTrs \Rightarrow\ (s\ \mathcal{CD}\ t_i))$.

Assume that a *compensatable component* of s is a component of s which can commit its operations even before s commits, but if s subsequently aborts, the compensating transaction $comp\_t_i$ of the committed component t_i must commit. Compensatable components are members of the set *Comp_Trs*.

$Abort_s \in H \Rightarrow \forall 0 \leq i \leq n\ (t_i \in Comp\_Trs \Rightarrow (comp\_t_i\ \mathcal{SCD}\ t_i))$.

Recall that $\mathcal{SCD}$ stands for strong commit dependency whereby if t' commits, t'' must commit.

Compensating transactions need to observe a state consistent with the effects of their corresponding components and hence, compensating transactions must execute (and commit) in the reverse order of the commitment of their corresponding components. We can capture this requirement by imposing a *begin-on-commit* $\mathcal{BCD}$ dependency on compensating transactions:

$\forall t_i\ tj \in Comp\_Trs\ ((Commit_{t_i} \rightarrow Commit_{t_j}) \Rightarrow (comp\_t_i\ \mathcal{BCD}\ comp\_t_j))$.

Begin-on-commit dependency states that transaction t_j cannot begin executing until transaction t_i has committed.

$(t_j\ \mathcal{BCD}\ t_i)$: $(Begin_{t_j} \in H \Rightarrow (Commit_{t_i} \rightarrow Begin_{t_j}))$

Suppose we assume that a compensating transaction compensates the effects of a component by invoking the *undo* operations of each of the operations invoked by the component. In this case, the compensating transaction must be allowed to view (the current state of) only those objects accessed by the corresponding component:

$View_{comp\_t_i} = Projection(H_{ct}, (\forall t, ob, p\ p_t[ob] \mid \exists q\ q_{t_i}[ob] \in H_{ct}))$.

Since we assume that all component transactions, including non-compensatable ones, can commit at any time, non-compensatable components should not be allowed to commit their effects on objects when they commit. There are a number of ways to structure non-compensatable component transactions [6, 9]. The simplest method is to structure them as subtransactions (as in nested transactions) which at commit time delegate all the operations in their AccessSet to s.

$\forall 0 \leq i \leq n$

$(t_i \notin Comp\_Trs \Rightarrow (\mathsf{Commit}_{t_i} \in H \Leftrightarrow Delegate_{t_i}[s, AccessSet_{t_i}] \in H))$.

It is possible to continue the development of our simple hierarchical transaction

model but at this point we have already considered all the basic interactions among the various special component transactions. For instance, it is possible to require that some component transactions execute in a predefined order as in the case of the Saga transaction model [12].

4.3.2 Complete Specification

Now let us put everything together. These axioms constitute the specifications of the open nested transaction model.

DEFINITION 4.12: *Axiomatic definition of Open Nested Transactions*
s denotes a top-level transaction.
t_a denotes either a top-level or a component transaction.
t_c denotes a compensatable component. $t_c \in Comp\_Trs$
$comp\_t_c$ denotes a compensating transaction of t_c.
t_p denotes a transaction which is not a non-compensatable component.

$$t_p = s \vee t_p = Comp\_Trs \vee t_p = comp\_t_c$$

t denotes either a top-level, a component, or a compensating transaction.

1. $SE_t = \{\mathsf{Begin}, \mathsf{Commit}, \mathsf{Abort}\}$
2. $IE_t = \{\mathsf{Begin}\}$
3. $TE_t = \{\mathsf{Commit}, \mathsf{Abort}\}$
4. t satisfies the fundamental Axioms I to IV
5. $View_{t_a} = H_{ct}$.
6. $ConflictSet_t = \{p_{t'}[ob] \mid ResponsibleTr(p_{t'}[ob]) \neq t, \mathcal{I}nprogress(p_{t'}[ob])\}$
7. $\forall ob\ \exists p\ p_t[ob] \in H \Rightarrow (ob\ is\ atomic)$
8. $\mathsf{Commit}_t \in H \Rightarrow \neg(t\ \mathcal{C}_N^*\ t)$
9. $\exists ob\ \exists p\ \exists t'\ Commit_{t_p}[p_{t'}[ob]] \in H \Rightarrow \mathsf{Commit}_{t_p} \in H$
10. $\mathsf{Commit}_{t_p} \in H \Rightarrow \forall ob\ \forall p\ \forall t'\ (p_{t'}[ob] \in AccessSet_{t_p} \Rightarrow Commit_{t_p}[p_{t'}[ob]] \in H)$
11. $\exists ob\ \exists p\ \exists t'\ Abort_t[p_{t'}[ob]] \in H \Rightarrow \mathsf{Abort}_t \in H$
12. $\mathsf{Abort}_t \in H \Rightarrow \forall ob\ \forall p\ \forall t'\ (p_{t'}[ob] \in AccessSet_t \Rightarrow Abort_t[p_{t'}[ob]] \in H)$
13. $View_{comp\_t_c} = Projection(H_{ct}, (\forall t', ob, p\ p_{t'}[ob] \mid \exists q\ q_{t_c}[ob] \in H_{ct}))$
14. $\forall t\ \notin Comp\_Trs\ \mathsf{Commit}_t \in H \Leftrightarrow Delegate_t[s, AccessSet_t] \in H$
15. $Begin_t \in H \Rightarrow ((t\ \mathcal{WD}\ s) \wedge$
    $(t \in VitalTrs \Rightarrow (s\ \mathcal{AD}\ t)) \wedge (t \notin VitalTrs \Rightarrow\ (s\ \mathcal{CD}\ t))).$
16. $Abort_s \in H \Rightarrow \forall i\ (t_i \in Comp\_Trs \Rightarrow (comp\_t_i\ \mathcal{SCD}\ t_i))$
17. $\forall t_i\ t_j \in Comp\_Trs\ ((Commit_{t_i} \rightarrow Commit_{t_j}) \Rightarrow (comp\_t_i\ \mathcal{BCD}\ comp\_t_j))$

In summary, Axioms 1 to 12 are similar to the corresponding ones of atomic transactions. All twelve axioms pertain to top-level transactions and their compensatable components.

As in the case of atomic transactions, everything is visible to these transactions [Axiom 5] whereas only objects accessed by a component are visible to its compensating transaction [Axiom 13].

For all transactions, a transaction's operations conflict with all ongoing operations invoked by other transactions [Axiom 6]. The serialization order must be acyclic, i.e., the transactions must be serializable taking into consideration the presence of delegation [Axiom 8].

Axioms 8 to 12 state the failure atomicity property of open nested transactions whereas Axioms 14 to 17 capture their failure properties with respect to compensatable and non-compensatable transactions. When a non-compensatable component commits, it delegates its access set to its top-level transaction [Axiom 14]. If top-level transaction aborts the compensating transaction $comp\_t_i$ for the committed component t_i must commit [Axiom 16]. Compensating transactions must execute (and commit) in the reverse order of the commitment of their corresponding components [Axiom 17].

Axiom 15 states that when a component begins, the component has a weak-abort dependency on its top-level transaction; also, if the component is vital, the top-level transaction has an abort dependency on the component, otherwise the top-level transaction has a commit dependency on the component.

The synthesis process followed above can be viewed as the derivation of a new model by combining and modifying the specifications of existing transaction models, namely, nested transactions and sagas [12]. We should point out that our open nested model is similar to the DOM transactions [3].

Obviously, the nested transaction model and the open nested transaction model have different properties merely due to the fact that they involve different types of component transactions. (Subtransactions of nested transactions are non-vital and non-compensatable.) This is still the case even if we consider the special case of an open nested transaction all of whose component transactions are non-vital and non-compensatable and compare it with a two-level nested transaction. The reason is that these two special nested and open nested transactions have different concurrent behaviors and different visibility properties because of the differences in the specifications of views and conflict sets. But for these differences, the two special cases of nested and open nested transactions have the same permanence and recovery properties since (1) they have similar structure-induced dependencies, and (2) their *Commit* and *Abort* events have similar semantics.

The above exercise reveals the many advantages of using a simple formalism like ACTA to deal with extended transactions: We can precisely state the behavior of transactions adhering to a given transaction model; We can analyze if higher-level requirements are satisfied; We can modify some of the properties to tailor a different transaction

model; We can precisely delineate the differences between models and understand what contributes to the differences and similarities between transaction models.

5 Conclusion

The ACTA transaction framework used in this paper to synthesize new transaction models was motivated by a need to provide a RISC-like meta model for treating extended transactions. We showed how the building blocks of ACTA, namely, history, dependencies, view of a transaction, and conflict set of a transaction along with the notion of delegation, glued by first-order logic, serve as powerful tools for the development of new transaction models in a systematic and precise way.

We also showed how ACTA is amenable to the synthesis of new transaction models by tailoring existing models or by starting from first principles. Specifically, chain transactions were a result of a restriction imposed on the invocation of the Join event associated with joint transactions such that they result in linear structures only. This restriction was captured by an axiom which, when added to the axiomatic definition of joint transactions, yields the definition of chain transactions. Also, reporting transactions and co-transactions were derived from joint transactions by removing the restriction that Join be a terminating event. This allows a transaction to join multiple times with another transaction, thereby delegating more operations to the joint transaction. Co-transactions are more flexible than reporting transactions since they allow transactions to join back and forth.

Nested-split transactions were derived by combining the axiomatic definitions of nested transactions and split transactions, the requirement being that nested-split transactions retain the properties of nested and split transactions.

Finally, an open nested transaction model was synthesized starting from first principles. The behavior of transactions adhering to the model were derived from high-level requirements.

ACTA has also been applied to derive from the original definition of the saga model [12], more flexible saga models in which failed components can be retried, replaced with alternative ones, or ignored. More flexibility was achieved by introducing new component transaction types, new significant events associated with these types, and new dependencies describing the relationship between these new transaction types [9].

These new extended transactions models show that besides supporting the specification and analysis of existing transaction models [4, 8, 6], ACTA has the power to support the synthesis and analysis of new extended transaction models in a systematic way, as well as the tailoring of existing ones. Thus, handling the requirements of new database

applications is facilitated.

References

[1] Badrinath, B. and Ramamritham, K. Semantics-based concurrency control: Beyond Commutativity. *ACM Transactions on Database Systems*, March 1992.

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

[3] Buchmann, A. et al. A Transaction Model for Active Distributed Object Systems. In Elmagarmid, A. K., editor, *Database Transaction Models for Advanced Applications*, pages 123–158. Morgan Kaufmann, 1992.

[4] 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.

[5] Chrysanthis, P. K. and Ramamritham, K. A Unifying Framework for Transactions in Competitive and Cooperative Environments. *IEEE Bulletin on Office and Knowledge Engineering*, 4(1):3–21, February 1991.

[6] 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.

[7] Chrysanthis, P. K., Raghuram, S., and Ramamritham, K. Extracting Concurrency from Objects: A Methodology. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 108–117, May 1991.

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

[9] Chrysanthis, P. K. and Ramamritham, K. ACTA: The SAGA Continues. In Elmagarmid, A. K., editor, *Database Transaction Models for Advanced Applications*, pages 349–398. Morgan Kaufmann, 1992.

[10] Du, W. and Elmagarmid, A. K. Quasi Serializability: a Correctness Criterion for Global Concurrency Control in InterBase. In *Proceedings of the Fifteenth International Conference on Very Large Databases*, pages 347–355, August 1989.

[11] 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.

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

[13] Herlihy, M. P. and Weihl, W. Hybrid concurrency control for abstract data types. In *Proceedings of the Seventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, Austin, Texas*, pages 201–210, March 1988.

[14] 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.

[15] Mohan, C., Haderle, D., Lindsat, B., Pirahesh, H., and Schwarz, P. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *(to appear) in ACM Transactions on Database Systems (Also Available as IBM Computer Science Research Report RJ 6649)*, January 1989.

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

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

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

[19] Skarra, A. Localized Correctness Specifications for Cooperating Transactions in an Object-Oriented Database. *IEEE Bulletin on Office and Knowledge Engineering*, 4(1):79–106, February 1991.

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

[21] Weihl, W. Commutativity-Based concurrency control for abstract data types. *IEEE Transactions on Computers*, 37(12):1488–1505, December 1988.