Synthesis of Extended Transaction Models Using ACTA

PANOS K. CHRYSANTHIS
University of Pittsburgh
and
KRITHI RAMAMRITHAM
University of Massachusetts

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 article 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.

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]. Distributed Systems—*distributed databases*: D.2.4 [**Software Engineering**]: Program Verification —*correctness proofs*; D.3 3 [**Programming Languages**]: Language Constructs and Features—*abstract data types*; D.4.1 [**Operating Systems**]: Process Management—*concurrency*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*assertions*; H.2.4 [**Database Management**]: Systems—*concurrency*; *distributed systems*; *transaction processing*

General Terms: Design, Reliability, Theory, Verification

Additional Key Words and Phrases: Concurrency control, correctness criteria, semantics, serializability theory, transaction models

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

Authors' addresses: P K. Chrysanthis, Department of Computer Science, University of Pittsburgh, Pittsburgh, PA 15260; K. Ramamritham, Department of Computer Science, University of Massachusetts, Amherst, MA 01003.

1. INTRODUCTION

Although powerful, the transaction model adopted in traditional database systems lacks 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,[1] 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*, *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 to be considered.

In Chrysanthis [1991] and Chrysanthis and Ramamritham [1991b], 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 article presents ACTA *as a tool for the synthesis of extended transaction models*, one that 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 3.1.2), *Reporting transactions* (Section 3.1.3), and *Cotransactions* (Section 3.1.4) by modifying the specification of joint transactions [Pu et al. 1988], and derive the *Nested-Split-transaction* model (Section 3.2) by combining the specifications of nested- and split-transaction models [Moss 1981; Pu et al. 1988]. As an example of the latter, we synthesize in Section 3.3 an *open-nested-transaction* model from the high-level requirements on transactions adhering to the model.

[1]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.

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 their effect on objects by means of constraints on histories. Intertransaction dependencies, discussed in Section 2.3, form 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 operations are atomic and 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 2.1.1.1. Invocation of an operation of 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[2] by transaction t (i.e., $p_t[ob] \in OE_t$).

The effects of an operation p invoked by a transaction t on an object ob are 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 corresponding event is denoted by $Commit[p_t[ob]]$.

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

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

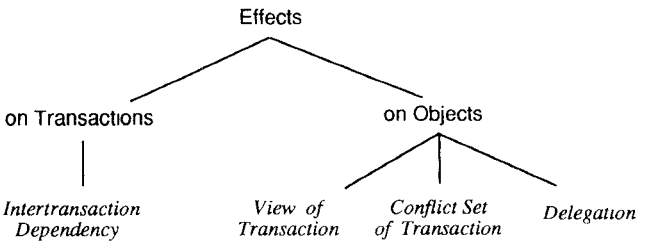

Fig. 1. Dimensions of the ACTA framework.

denoted by $Abort[p_t[ob]]$. Thus, once an operation is aborted, the *state* of the object will be as though the aborted operation never executed.

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. For instance, in the case of atomic objects assumed by most transaction models, all operations that have observed the effects of an aborted operation are also aborted. For example, if the return values of subsequently executed operations reflect the state of the object produced by the aborted operation, these operations would also be aborted in the case of atomic objects (see Section 2.5.1). Nonatomic objects, on the other hand, permit weaker consistency and recovery semantics such that operations that observed the effects of aborted operations may not be affected.

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.[3] Other transaction manage-

[3]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.

ment primitives include Spawn, found in the nested-transaction model, and Split and Join, found in the split-transaction model [Pu et al. 1988].

Definition 2.1.2.1. 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 2.1.2.2. *Initiation events*, denoted by IE_t, are a set of significant events that can be invoked to initiate the execution of transaction t: $IE_t \subset SE_t$.

Definition 2.1.2.3. *Termination events*, denoted by TE_t, are 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 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* [Bernstein et al. 1987] which represents the concurrent execution of a set of transactions T. ACTA captures both the effects of transactions on other transactions and 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 2.2.1. A *history* H of the concurrent execution of a set of transactions T contains all the events, significant events, and object events invoked by the transactions in T and indicates the (partial) order in which these events occur.

Definition 2.2.2. 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 *projection* of a

history H according to a given criterion is a subhistory that satisfies the criterion. For instance, the projection of a history with respect to committed transactions, denoted by H_{comm}, includes only those events invoked by committed transactions. 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 2.2.3. $(\epsilon \in H) \Rightarrow Condition_H$, 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 2.2.4. $Condition_H \Rightarrow (\epsilon \in H)$ 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 constraints 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.

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 $\mathscr{CD}$ 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.,

$$\mathsf{Commit}_{t_j} \in H \Rightarrow \left(\mathsf{Commit}_{t_i} \in H \Rightarrow \left(\mathsf{Commit}_{t_i} \rightarrow \mathsf{Commit}_{t_j}\right)\right).$$

Abort Dependency ($t_j \mathscr{AD}\ t_i$). 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 ($t_j \mathscr{WD}\ t_i$). 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.,

$$\mathsf{Abort}_{t_i} \in H \Rightarrow \left(\neg \left(\mathsf{Commit}_{t_j} \rightarrow \mathsf{Abort}_{t_i} \right) \Rightarrow \left(\mathsf{Abort}_{t_i} \in H \right) \right).$$

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 Chrysanthis [1991]). In this sense, ACTA is an open-ended framework.

Besides the logical representation introduced above, intertransaction 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 3.2. 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 intertransaction dependencies occur.

2.3.2 *Source of Dependencies.* Dependencies between transactions may be a direct result of the structural properties of transactions, or may develop indirectly 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 ($t_c \mathscr{WD}\ t_p$) and a parent establishing a commit dependency on its child ($t_p \mathscr{CD}\ t_c$).

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

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

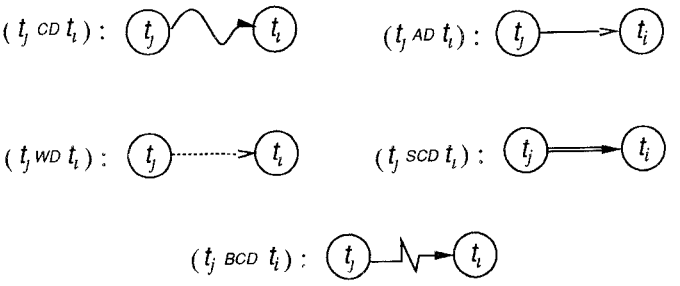

Fig. 2. Intertransaction dependencies graph.

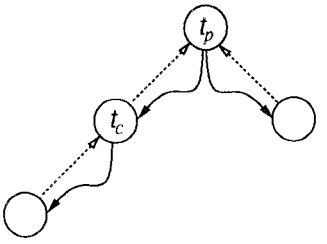

Fig. 3. Structure of nested transactions.

permanent in the database. They are just made visible to its parent. See Section 3.2.1 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, we can say that two operations conflict if the order of their execution matters. For example, in the traditional framework, a compatibility table [Bernstein et al. 1987] of an object *ob* expresses simple relations between conflicting operations. A conflict relation has the form

$$\left(p_{t_i}[ob] \rightarrow q_{t_j}[ob]\right) \Rightarrow (t_j\ \mathscr{D}\ t_i)$$

indicating that if transaction t_i 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 $\mathscr{D}$ 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

Correctness of concurrent transaction executions depends on how transactions affect each other as well as how they affect the objects. This, in turn, depends on the effects of the significant events associated with a transaction

and of the operations invoked by a transaction. We studied the former in the previous subsection. We focus on the latter now.

We begin with a discussion of the effects of operations on transactions and their interrelationships. The visibility of the effects of one transaction's operations to another transaction are then discussed.

2.4.1 *Conflicts between Operations and the Induced Dependencies.* We begin with the notion of conflicts between operations on an object and discuss how it induces dependencies between transactions. We then refine it into return-value-dependent and return-value-independent conflicts so as to weaken the induced dependencies. Section 2.5 uses these notions to define formally the correctness of both transactions and objects.

$H^{(ob)}$, the projection of the history H with respect to an object ob, contains the history of operation invocations on an object ob. $H^{(ob)} = p_1 \circ p_2 \circ \cdots \circ p_n$ 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 2.4.1.1. Two operations p and q *conflict* in a state produced by $H^{(ob)}$, denoted by $conflict(H^{(ob)}, p, q)$, iff

$$(state(H^{(ob)} \circ p, q) \neq state(H^{(ob)} \circ q, p)) \vee$$
$$(return(H^{(ob)}, q) \neq return(H^{(ob)} \circ p, q)) \vee$$
$$(return(H^{(ob)}, p) \neq return(H^{(ob)} \circ q, p)).$$

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.

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 arguments in *conflict* 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 both 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 employs 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 article, we will consider the situation when the effects are immediate. The effects of considering deferred updates are considered in Section 3.4.

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

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

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:

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

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.[4]

Since state changes are observed only via return values, the return values of conflicting operations can be considered to produce weaker types of dependencies than abort dependencies. Toward this end, it is useful to distinguish between return-value-dependent and return-value-independent conflicts.

Definition 2.4.1.2. *return-value-independent*$(H^{(ob)}, p, q)$ is true if *conflict*$(H^{(ob)}, p, q)$ is true and the return value of q is independent of whether p precedes q, i.e., $return(H^{(ob)} \circ p, q) = return(H^{(ob)}, q)$; *return-value-dependent*$(H^{(ob)}, p, q)$ is true if *conflict*$(H^{(ob)}, p, q)$ is true and $return(H^{(ob)} \circ p, q) \neq return(H^{(ob)}, q)$.

Whereas commutativity does not distinguish between return-value-dependent and return-value-independent conflicts, a weaker conflict notion, called *recoverability* [Badrinath and Ramamritham 1992] results if we do. Specifically, the weaker $\mathscr{CD}$ relationship is induced between return-value-independent conflicting operations rather than $\mathscr{AD}$:

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

The generality of the conflict relations allows ACTA to capture different types of type-specific concurrency control discussed in the literature [Badrinath and Ramamritham 1992; Chrysanthis et al. 1991; Herlihy and

[4]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*.

Weihl 1988; Schwarz and Spector 1984; Weihl 1988], and even to tailor them for cooperative environments [Fernandez and Zdonik 1989; Skarra 1991].

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 2.4.2.1. 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. Additionally, 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 criterion, *Projection_Condition*, typically involving H_{ct}, the current history. In other words, $View_t$ is the subhistory constructed by eliminating any events in H_{ct} that do not satisfy the given *Projection_Condition* 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 transaction t_p. The *Projection_Condition* used to construct the view of such a subtransaction t_c is specified as follows.

$$\forall q, t, ob, q_t[ob] \in View_{t_c} \Rightarrow \exists r\, r_{t_p}[ob] \in H_{ct}$$

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, the parent of t_c, has performed some operation r.

Conflict Set of a Transaction

Definition 2.4.2.2. The *conflict set* of a transaction t, denoted by $ConflictSet_t$, contains those operations in the current history with respect to which the effects of conflicts have to be determined when t invokes an operation.

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

$ConflictSet_t$ is a subset of the object events in H_{ct} that satisfy some *Predicate*:

$$ConflictSet_t = \{p_{t_i}[ob] \mid Predicate\}.$$

For example, let us consider nested transactions once again. In nested transactions, a subtransaction t_c can access any object accessed currently by one of its ancestors t_a, even if the operations performed by t_c conflict with those performed by t_a. This is captured by:

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

In other words, any operation invoked by an ancestor of t_c is not contained in $ConflictSet_{t_c}$. That is, the effects of the conflict between an operation invoked by a child transaction t_c and an operation p invoked by t_i on an object ob need to be taken into account only if (1) t_i and t_c are different, (2) t_i is not an ancestor (in the nested-transaction structure) of t_c, and (3) p is still in progress. For this reason, a transaction t_c can invoke an operation that conflicts with another in progress, invoked by its ancestor t_a, ignoring the dependencies that may form in the process.

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 $View_t$ 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 specify also 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 2.4.2.3. $ResponsibleTr(p_{t_i}[ob])$ 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 2.4.2.4. $Delegate_{t_i}[t_j, p_{t_i}[ob]]$ 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 Chrysanthis [1991]:

—$ResponsibleTr(p_{t_i}[ob])$ 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 effect if the delegated operations have already been committed or aborted.

—From the perspective 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—the 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 2.4.2.5. $AccessSet_t = \{p_{t_i}[ob] \mid ResponsibleTr(p_{t_i}[ob]) = t\}$; 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 Cotransactions and Reporting Transactions, described in Section 3.

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 while ignoring the effects of conflicts, and by delegating operations to each other. By being able to capture these aspects of transactions, the ACTA framework is applicable to cooperative environments.

In the rest of the article, we assume that delegation is done *by the system* in response to the invocation of a transaction management event, such as Commit in the above example. This implies that as far as the history is concerned, the commit and delegate events occur concurrently.

2.5 Simple Examples of ACTA Specifications

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 transactions and objects starting with the serializability correctness criterion and the failure atomicity property. Subsequently, we state a set of axioms that are applicable to all transaction models.

2.5.1 *Serializability, Object Correctness, and Failure Atomicity*

Let T be the set of transactions.
Let $\mathscr{C}$ be a binary relation on transactions in T.
Let T_{comm} be the subset of T containing committed transactions.
Let H_{comm} be the history of events relating to transactions in T_{comm}.

Definition 2.5.1.1 *Serializability*

$\forall t_i, t_j \in T_{comm}, t_i \neq t_j$

$$(t_i \mathscr{C} t_j) \text{ iff } \exists ob\ \exists p, q\Big(conflict\big(p_{t_i}[ob], q_{t_j}[ob]\big) \wedge \big(p_{t_i}[ob] \rightarrow q_{t_j}[ob]\big)\Big)$$

Let $\mathscr{C}^*$ be the transitive-closure of $\mathscr{C}$; i.e.,

$$(t_i \mathscr{C}^* t_k) \text{ iff } \big[(t_i \mathscr{C} t_k) \vee \exists t_j (t_i \mathscr{C} t_j \wedge t_j \mathscr{C}^* t_k)\big].$$

H_{comm} is (*conflict*) *serializable* iff $\forall t \in T_{comm}\ \neg(t \mathscr{C}^* t)$.

Conflicting operations induce serialization ordering requirements (denoted by the $\mathscr{C}$ relation above), and serializability demands that this ordering must be acyclic. Whereas serializability is concerned with the correctness of execution of committed transactions, we must also worry about the correctness of the objects as operations execute, and more importantly, as operations abort. First, we must ensure that operations on individual objects also execute serializably, that is, as if the committed transactions visited the objects one after another. Second, we must ensure that when an operation aborts, it also

aborts any other operation which is return-value dependent on it and therefore has observed the effects of the aborted operation.

Definition 2.5.1.2. *Objects' Correctness.* An object *ob* behaves *correctly* iff

$$\forall t_i t_j \in T, t_i \neq t_j, \forall p, q$$

$$\Big(\textit{return-value-dependent}(p, q) \wedge \big(p_{t_i}[ob] \rightarrow q_{t_j}[ob]\big)\Big) \wedge$$

$$\neg\Big(\big(\textit{Commit}\big[p_{t_i}[ob]\big] \rightarrow q_{t_j}[ob]\big) \vee \big(\textit{Abort}\big[p_{t_i}[ob]\big] \rightarrow q_{t_j}[ob]\big)\Big) \Rightarrow$$

$$\Big(\big(\textit{Abort}\big[p_{t_i}[ob]\big] \in H^{(ob)}\big) \Rightarrow \big(\textit{Abort}\big[q_{t_j}[ob]\big] \in H^{(ob)}\big)\Big).$$

$$\forall t_i, t_j \in T_{comm}, t_i \neq t_j$$

$$(t_i \mathscr{C}_{ob} t_j) \text{ iff } \exists p, q\Big(\textit{conflict}\big(p_{t_i}[ob], q_{t_j}[ob]\big) \wedge \big(p_{t_i}[ob] \rightarrow q_{t_j}[ob]\big)\Big)$$

An object *ob* behaves *serializably* iff

$$\forall t \in T_{com} \neg(t \mathscr{C}^{*}_{ob} t).$$

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

The first statement states that for an object to behave *correctly* it must ensure that when an operation aborts, any return-value-dependent operation that is subsequently executed, but prior to the abortion, must also be aborted. This ensures the correct behavior of objects in the presence of failures assuming immediate effects of operations on objects. A serializable behavior of an object is ensured by preventing committed transactions from forming cyclic $\mathscr{C}_{ob}$ relationships where $\mathscr{C}_{ob}$ considers only the $\mathscr{C}$ relationships that occur from accessing the object *ob* ($\mathscr{C}_{ob} \subseteq \mathscr{C}$).

Definition 2.5.1.3. Transaction t is *failure atomic* if

(1) $\exists ob\ \exists p\ (\textit{Commit}[p_t[ob]] \in H) \Rightarrow$
$\forall ob'\ \forall q\ ((q_t[ob'] \in H) \Rightarrow (\textit{Commit}[q_t[ob']] \in H))$

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

As mentioned earlier, failure atomicity implies that all or none of a transaction's operations are committed (by some transaction). 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 [Du and Elmagarmid 1989] and *predicatewise serializability* [Korth and Speegle 1988] can be expressed in ACTA [Chrysanthis 1991].

2.5.2 *Fundamental Axioms of Transactions.* 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 t is always associated with a set of initiation-significant events (IE_t) that can be invoked to initiate the execution of the transaction, and a set of termination-significant events (TE_t) 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 2.5.2.1. *Fundamental Axioms of Transactions.* Let t be a transaction and H^t the projection of the history H with respect to t.

(1) $\forall \alpha \in IE_t\ (\alpha \in H^t) \Rightarrow \nexists\ \beta \in IE_t\ (\alpha \rightarrow \beta)$
A transaction cannot be initiated by two different events.

(2) $\forall \delta \in TE_t\ \exists \alpha \in IE_t\ (\delta \in H^t) \Rightarrow (\alpha \rightarrow \delta)$
If a transaction has terminated, it must have been previously initiated.

(3) $\forall \gamma \in TE_t\ (\gamma \in H^t) \Rightarrow \nexists\ \delta \in TE_t\ (\gamma \rightarrow \delta)$
A transaction cannot be terminated by two different termination events.

(4) $\forall ob\ \forall p\ (p_t[ob] \in H) \Rightarrow ((\exists \alpha \in IE_t\ (\alpha \rightarrow p_t[ob])) \wedge (\exists \gamma \in TE_t\ (p_t[ob] \rightarrow \gamma)))$
Only in-progress transactions can invoke operations on objects.

2.5.3 *Axiomatic Definition of Atomic Transactions.* Below we express in ACTA the basic properties of atomic transactions with a set of axioms.

Definition 2.5.3.1. *Axiomatic Definition of Atomic Transactions.* t denotes an atomic transaction.

(1) SE_t = {Begin, Commit, Abort}

(2) IE_t = {Begin}

(3) TE_t = {Commit, Abort}

(4) t satisfies the fundamental Axioms 1 to 4 (Definition 2.5.2.1)

(5) $View_t = H_{ct}$
A transaction sees the current state of the objects in the database.

(6) $ConflictSet_t = \{p_{t'}[ob] \mid ResponsibleTr(p_{t'}[ob]) \neq t, Inprogress(p_{t'}[ob])\}$
Effects of conflicts have to be considered against all in-progress operations performed by different transactions and for which t is not responsible.

(7) $\forall ob\ \exists p\ p_t[ob] \in H \Rightarrow (ob\ is\ atomic)$
All objects upon which an atomic transaction invokes an operation are atomic objects (see Definition 2.5.1.2). That is, they detect conflicts and induce the appropriate dependencies.

(8) $\mathsf{Commit}_t \in H \Rightarrow \neg(t\ \mathscr{C}^*\ t)$.
An atomic transaction can commit only if it is not part of a cycle of $\mathscr{C}$ relations developed through the invocation of conflicting operations.[5] This and the next two axioms define the semantics of the Commit event of atomic transactions in terms of the *Commit* operation defined on objects.

(9) $\exists ob\ \exists p\ Commit_t[\,p_t[ob]\,] \in H \Rightarrow \mathsf{Commit}_t \in H$
If an operation is committed on an object, the invoking transaction must commit.

(10) $\mathsf{Commit}_t \in H \Rightarrow \forall ob\ \forall p(\,p_t[ob] \in H \Rightarrow Commit_t[\,p_t[ob]\,] \in H)$
If a transaction commits, all the operations invoked by the transaction are committed.

(11) $\exists ob\ \exists p\ Abort_t[\,p_t[ob]\,] \in H \Rightarrow \mathsf{Abort}_t \in H$
If an operation is aborted on an object, the invoking transaction must abort. This and the following axiom define the semantics of the Abort event in terms of the *Abort* operation defined on objects.

(12) $\mathsf{Abort}_t \in H \Rightarrow \forall ob\ \forall p(\,p_t[ob] \in H \Rightarrow Abort_t[\,p_t[ob]\,] \in H)$
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 Chrysanthis [1991]).

3. SYNTHESIZING NEW TRANSACTION MODELS

Below we synthesize two new families of extended-transaction models. The first is derived from the *joint-transaction* model [Pu et al. 1988]. The second is derived from the *nested-transaction* model [Moss 1981] and the *split-transaction* model [Pu et al. 1988]. 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 model is that they support *delegation* between transactions. The following definition of conflicts takes into account the presence of delegation.

Definition 3.1. Let $\mathscr{C}_N$ be a binary relation on transaction in T_{comm}.

$$\forall t_i, t_j \in T_{comm}, t_i \neq t_j$$
$$(t_i\ \mathscr{C}_N\ t_j) \text{ iff}$$
$$\exists ob\ \exists p, q\ \exists t_m, t_n \left(conflict(\,p_{t_m}[ob], q_{t_n}[ob]\,)\right.$$
$$\wedge\left(\,p_{t_m}[ob] \rightarrow q_{t_n}[ob]\,\right) \wedge$$
$$\left(ResponsibleTr(\,p_{t_m}[ob]\,) = t_i\right)$$
$$\left.\wedge\left(ResponsibleTr(\,p_{t_n}[ob]\,) = t_j\right)\right)$$

[5] Note that the atomicity property local to individual objects is not sufficient to guarantee serializable execution of concurrent transactions across all objects [Weihl 1984].

This definition extends the definition of the $\mathscr{C}$ relation (Definition 2.5.1.1) to include the serialization orderings due to the delegated objects. To see that $\mathscr{C}_N$ is a generalization of $\mathscr{C}$ $((t_i \mathscr{C} t_j) \Rightarrow (t_i \mathscr{C}_N t_j))$, 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 $\mathscr{C}_N$ for $\mathscr{C}$ in the definition of serializability (Definition 2.5.1.1), transactions are serialized with respect to operations for which they are responsible.

Definition 3.2. H_{comm} is (*conflict*) *serializable* iff

$$\forall t \in T_{comm} \neg (t \mathscr{C}_N^* t).$$

There is no need to revisit the definition of failure atomicity in light of delegation. Failure atomicity does not require the invoking transaction of an operation to be the transaction to commit or abort the operation. Thus, failure atomicity (Definition 2.5.1.3) 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 3.3. Transaction t is *quasi-failure-atomic* iff

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

(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 that 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 general, in the absence of delegation, quasi-failure-atomicity is equivalent to failure atomicity. More specifically, if delegation does not occur from a transaction, its being quasi-failure-atomic implies that it is failure atomic.

3.1 Joint-Transaction Model and its Variations

In this section, we derive three new extended-transaction models, namely, *chain transactions*, *reporting transactions*, and *cotransactions*, though a series of manipulations, beginning with the axiomatic definition of joint transactions [Pu et al. 1988]. In Chrysanthis and Ramamritham [1991a], we defined these models using *dependency production rules*, a formalism close to dependency graphs which captures 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.

3.1.1 *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 3.1.1.1. *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 = {Begin, Join, Commit, Abort}
(2) IE_t = {Begin}
(3) TE_t = {Join, Commit, Abort}
(4) t satisfies the fundamental Axioms 1 to 4 (Definition 2.5.2.1)
(5) $View_t = H_{ct}$
(6) $ConflictSet_t = \{p_{t'}[ob] \mid ResponsibleTr(p_{t'}[ob]) \neq t,\ Inprogress(p_{t'}[ob])\}$
(7) $\forall ob\ \exists p\ p_t[ob] \in H \Rightarrow (ob\ is\ atomic)$
(8) $\text{Commit}_t \in H \Rightarrow \neg(t\ \mathscr{C}_N^*\ t)$
(9) $\exists ob\ \exists q\ \exists t_i\ Commit_t[q_{t_i}[ob]] \in H \Rightarrow \text{Commit}_t \in H$
(10) $\text{Commit}_t \in H \Rightarrow \forall ob\ \forall q\ \forall t_i(q_{t_i}[ob] \in AccessSet_t \Rightarrow Commit_t[q_{t_i}[ob]] \in H)$
(11) $\exists ob\ \exists q\ \exists t_i\ Abort_t[q_{t_i}[ob]] \in H \Rightarrow \text{Abort}_t \in H$
(12) $\text{Abort}_t \in H \Rightarrow \forall ob\ \forall q\ \forall t_i(q_{t_i}[ob] \in AccessSet_t \Rightarrow Abort_t[q_{t_i}[ob]] \in H)$
(13) $\text{Join}_{t_a}[t_b] \in H \Leftrightarrow Delegate_{t_a}[t_b, AccessSet_{t_a}] \in H$

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 a child transaction in the nested-transaction model when the child transaction commits (see Section 3.2.1).

In Axiom 13, the joint transaction is the only parameter of Join; however, as we will see below, an additional parameter needs to be associated with the Join event when deriving reporting transactions from joint transactions.

We now state some of the failure and ordering properties of joint transactions. Their proof can be found in Chrysanthis [1991].

LEMMA 3.1.1.2. *A transaction t in the joint-transaction model is quasi-failure-atomic.*

LEMMA 3.1.1.3. *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 joined by any other transaction.*

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

THEOREM 3.1.1.4. *A joining transaction t_a is serializable with respect to the joint transaction t_b iff* $\mathsf{Join}_{t_a}[t_b] \in H \Rightarrow \neg((t_a \; \mathscr{C}_N^* \; t_b) \wedge (t_b \; \mathscr{C}_N^* \; t_a))$.

This theorem states that if there is no cycle involving t_a and t_b then they are serializable.

COROLLARY 3.1.1.5. *A joining transaction t_a may not be serializable with respect to the joint transaction t_b.*

3.1.2 *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 3.1.2.1. *Axiomatic Definition of Chain Transactions*
t_a denotes a joining transaction.
t_b denotes a joint transaction.
t_b denotes either a joining or a joint transaction.

(1..13) Axiom 1..13 of Definition 3.1.1.1.

(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 3.1.1) 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.

3.1.3 *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 periodically *reports* its results to the joint transaction by delegating more operations to the joint

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

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 3.1.3.1. *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 = {Begin, Join, Commit, Abort}

(2) IE_t = {Begin}

(3) TE_t = {Commit, Abort}

(4..12) Axiom 4..12 of Definition 3.1.1.1

(13) $\text{Join}_{t_a}[t_b, ReportSet_{t_a}] \in H \Leftrightarrow Delegate_{t_a}[t_b, ReportSet_{t_a}] \in H$, $ReportSet_{t_a} \subseteq AccessSet_{t_a}$

(14) $\text{Join}_{t_a}[t_b, ReportSet_{t_a}] \in H \Rightarrow (t_a \mathscr{AD} t_b)$

(15) $\text{Join}_{t_a}[t_b, ReportSet_{t_a}] \in H \Rightarrow \nexists\, t,\ t \neq t_b\ (\text{Join}_{t_a}[t, ReportSet_{t_a}] \rightarrow \text{Join}_{t_a}[t_b, ReportSet_{t_a}])$

(16) $\text{Join}_{t_a}[t_b, ReportSet_{t_a}] \in H \Rightarrow \text{Join}_{t_b}[t_a, ReportSet_{t_b}] \notin H$

$ReportSet_{t_a}$ contains the operations on the objects to be delegated (Axiom 13). Since $ReportSet_{t_a} \subseteq AccessSet_{t_a}$, 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.

Note that the axioms do not prevent reporting transactions from forming nonlinear structures. If only linear structures must be permitted, Axiom 14 of chain transactions must be added to the above set of axioms. This point raises the issue of "completeness" of a set of axioms. We discuss this topic in Section 3.4.

Reporting transactions provide a more interesting control structure than joint transactions and can be useful in structuring data-driven computations. For example, consider a computation that requires remote access to a database over expensive communication links such as in a mobile computing environment. This computation can be split across the two sites using reporting transactions where the joining transaction executes in the database site whereas the joint transaction executes on the remote site. The joining transaction accesses the database and performs the initial processing on the data delegating to the joint transaction only those operations on data that need to be processed further at the remote site.

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 permitting a reporting transaction to join more than one transaction in which case they can support, for example, star-like computations.

3.1.4 *Cotransactions.* The characterization of reporting transactions allows t_a to continue its execution but prevents t_b from joining t_a. This is specified in Axiom 15 where $post(\epsilon)$ denotes the postcondition of event ϵ. Suppose that t_a is suspended when it joins t_b and 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*. The 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 *cotransactions* because they behave like *coroutines*, 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 cotransaction model specified below, the view of the cotransaction that resumes execution is restored to H_{ct}.

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

Here is the formal definition of cotransactions in ACTA:

Definition 3.1.4.1. *Axiomatic Definition of Cotransactions*
t_a denotes a joining transaction.
t_b denotes a joint transaction.
t denotes either a joining or a joint transaction.

(1) SE_t = {Begin, Join, Commit, Abort}
(2) IE_t = {Begin}
(3) TE_t = {Commit, Abort}
(4..14) Axiom 4..14 of Definition 3.1.3.1
(15) $post(\text{Join}_{t_a}[t_b]) \Rightarrow (View_{t_a} = \phi) \wedge (View_{t_b} = H_{ct})$
(16) $\text{Join}_{t_a}[t_b] \in H \Rightarrow (t_b \; \mathcal{SCD} \; t_a)$

Here $\mathcal{SCD}$ stands for *strong commit* dependency whereby if t_i commits, t_j must commit:

$$(t_j \; \mathcal{SCD} \; t_i): (Commit_{ti} \in H \Rightarrow Commit_{tj} \in H).$$

The termination semantics of cotransactions 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.

Cotransactions are useful in realizing applications that can be decomposed into interactive, and potentially distributed, subtasks which cannot execute in parallel. For instance, cotransactions can be used to set a meeting between two persons by having one cotransaction executing per person against the individual's calendar database. Cotransactions, as well as reporting transactions, can be easily modified to form more complex control structures in order to produce more interesting styles of cooperation.

3.2 Nested-Split-Transaction Model

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

In order to motivate the need for such a combined model, consider software development in which a developer structures her/his work in a hierarchical manner using nested transactions. Since software development may take an arbitrary long time, the designer would like to be able (1) to abort some of the operations of a nested transaction (subtransaction) when they are no longer needed, for example, after a failed attempt to fix a bug and (2) to split a long subtransaction into two sibling subtransactions which can commit or abort independently. Such requirements are not satisfied by either the nested or the split transaction models by themselves in an easy and straightforward manner but can be satisfied by a model that combines the properties of both.

3.2.1 *Nested Transactions.* In the nested-transaction model, e.g., Moss [1981], 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 parent. They can abort independently without causing the abortion of the whole transaction.

A subtransaction can access potentially any object that is currently accessed by one of its ancestor transactions. 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 3.2.1.1. *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 subtraction.

(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 1 to 4 (Definition 2.5.2.1)
(8) $View_{t_p} = H_{ct}$
(9) $ConflictSet_{t_0} = \{p_t[ob] \mid ResponsibleTr(p_t[ob]) \neq t_0,\ Inprogress(p_t[ob])\}$
(10) $\forall ob\ \exists p\ p_{t_p}[ob] \in H \Rightarrow (ob\ is\ atomic)$
(11) $\mathsf{Commit}_{t_p} \in H \Rightarrow \neg(t_p\ \mathscr{C}_N^*\ t_p)$
(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) $ConflictSet_{t_c} = \{p_t[ob] \mid ResponsibleTr(p_t[ob]) \neq t_c,\ t \notin Ancestors(t_c),$
$Inprogress(p_t[ob])\}$
(18) $\mathsf{Spawn}_{t_p}[t_c] \in H \Rightarrow Parent(t_c) = t_p$
(19) $\mathsf{Spawn}_{t_p}[t_c] \in H \Rightarrow (t_c\ \mathscr{WD}\ t_p) \wedge (t_p\ \mathscr{CD}\ t_c)$
(20) $\mathsf{Commit}_{t_c} \in H \Leftrightarrow Delegate_{t_c}[Parent(t_c),\ AccessSet_{t_c}] \in H$
(21) $\forall t \in Descendants(t_p)\ \forall ob\ \forall p,\ q(p_t[ob] \rightarrow q_{t_p}[ob])Conflict(p_t[ob],$
$q_{t_p}[ob]) \Rightarrow \exists t_c((Delegate_{t_c}[t_p,\ AccessSet_{t_c}] \rightarrow q_{t_p}[ob]) \wedge p_t[ob] \in$
$AccessSet_{t_c})$
(22) $Ancestor(t_c) = Ancestor(t_p) \cup \{t_p\} \wedge \forall t t_p \in Descendants(t) \Rightarrow$
$t_c \in Descendants(t)$

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, because of the *delegation* that occurs when a subtransaction commits, 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 defines the semantics of the Commit event of subtransactions, specifies clearly 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 depen-

dency $\mathscr{WD}$ and commit dependency $\mathscr{CD}$ 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 17 also states that operations delegated to the subtransaction and for which the subtransaction is responsible do not conflict with any operation invoked by the subtransaction.

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 ($t_p \mathscr{AD} t$) 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 recovery and concurrency properties of nested transactions can be shown (see Appendix and Chrysanthis [1991]). 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 may not be serializable.

3.2.2 *Split Transactions.* In the split-transaction model [Pu et al. 1988], 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 *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 3.2.2.1. *Axiomatic Definition of Split Transactions*

t_r denotes a primary transaction.
t_a denotes a splitting transaction, primary or split.

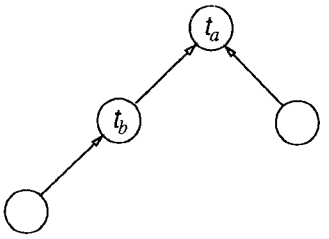

Fig. 4. Structure of split transactions.

t_b denotes the split transaction of t_a.
t denotes a transaction, primary or split.

(1) SE_{t_r} = {Begin, Split, Commit, Abort}
(2) IE_{t_r} = {Begin}
(3) TE_{t_r} = {Commit, Abort}
(4) SE_{t_b} = {Split, Commit, Abort}
(5) IE_{t_b} = {Split}
(6) TE_{t_b} = {Commit, Abort}
(7) t satisfies the fundamental Axioms 1 to 4 (Definition 2.5.2.1)
(8) $View_t = H_{ct}$
(9) $ConflictSet_{t_r} = \{p_t[ob] \mid ResponsibleTr(p_t[ob]) \neq t_r,\ Inprogress(p_t[ob])\}$
(10) $\forall ob\ \exists p\ p_t[ob] \in H \Rightarrow (ob\ is\ atomic)$
(11) $\text{Commit}_t \in H \Rightarrow \neg(t\ \mathscr{C}_N^*\ t)$
(12) $\exists ob\ \exists q\ \exists t_i\ Commit_t[q_{t_i}[ob]] \in H \Rightarrow \text{Commit}_t \in H$
(13) $\text{Commit}_t \in H \Rightarrow \forall ob\ \forall q\ \forall t_i\ (q_{t_i}[ob] \in AccessSet_t \Rightarrow Commit_t[q_{t_i}[ob]] \in H)$
(14) $\exists ob\ \exists q\ \exists t_i\ Abort_t[q_{t_i}[ob]] \in H \Rightarrow \text{Abort}_t \in H$
(15) $\text{Abort}_t \in H \Rightarrow \forall ob\ \forall q\ \forall t_i(q_{t_i}[ob] \in AccessSet_t \Rightarrow Abort_t[q_{t_i}[ob]] \in H)$
(16) $\text{Split}_{t_a}[t_b, CanAccess_{t_b}(t_a)] \in H \Rightarrow (CanAccess_{t_a}(t_a) \neq \phi \Rightarrow (t_b\ \mathscr{AD}\ t_a))$
(17) $\text{Split}_{t_a}[t_b, CanAccess_{t_b}(t_a)] \in H \Leftrightarrow Delegate_{t_a}[t_b, DelegateSet] \in H$
(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) $ConflictSet_{t_b} = \{p_t[ob] \mid (ResponsibleTr(p_t[ob]) \neq t_b,\ t \neq t_a,\ Inprogress(p_t[ob])) \vee (ResponsibleTr(p_t[ob]) = t_a,\ Inprogress(p_t[ob]) \wedge (p_t[ob] \notin CanAccess_{t_b}(t_a)))\}$
(20) $\forall ob\ \forall p,q(\exists r(r_{t_a}[ob] \in CanAccess_{t_b}(t_a))) \wedge p_{t_a}[ob] \in H \wedge Conflict(p_{t_a}[ob], q_{t_b}[ob] \Rightarrow (p_{t_a}[ob] \rightarrow q_{t_b}[ob])$

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).

Split$_{t_a}[t_b, CanAccess_{t_a}(t_a)]$ 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 a conflicting operation on them (Axiom 20).

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

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 are serializable, as shown in Chrysanthis [1991]. That is, if t_a splits t_b serially, then t_a precedes t_b in any serializable history in which both commit. If the split is independent then t_a and t_b in any serializable history in which both commit. If the split 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.

3.2.3 *Nested-Split Transactions.* Given our definitions for atomic transactions (see Definition 2.5.3.1), nested transactions (see Definition 3.2.1.1) and split transactions (see Definition 3.2.2.1) 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 transac-

[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.

tions). 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, e.g., 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 of Definition 3.2.1.1), the Split event induces an abort dependency of the split transaction on the splitting transaction (Axiom 19 of Definition 3.2.2.1). Additionally, in contrast to the Spawn event, due to delegation the Split event may associate a nonempty access set with the split transaction.

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.* In this model, given a nested transaction, it is possible to split the root or a subtransaction. A split transaction may further split creating another split transaction, or spawn a new subtransaction becoming a root of a new nested transaction. In this way, a set of possibly dependent nested transactions may be created (see Figure 5).

Definition 3.2.3.1. *Axiomatic Definition of Split-and-Nested Transactions*

t_0 denotes a root or a primary transaction. $Parent(t_0) = Ancestor(t_0) = \phi$.
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 3.2.1.1
(26..30) Axiom 16..20 of Definition 3.2.2.1
(31) $\text{Split}_{t_a}[t_b, CanAccess_{t_b}(t_a)] \in H \Rightarrow parent(t_b) = \phi$

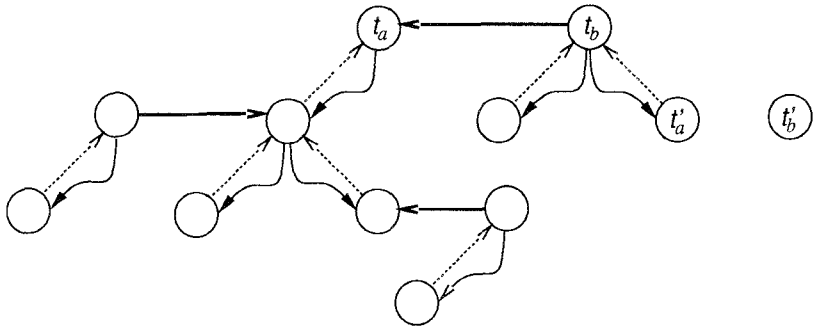

Fig. 5. Structure of split-and-nested transactions.

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 to the case of Begin and Spawn events (Axioms 19 and 21 (or Axioms 16 and 18 of nested transactions)).

The split-and-nested model produces only hierarchical transaction structures as in 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.

[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 operation, and (2) that if a root transaction aborts, all operations performed by the root and its descendants abort [Chrysanthis 1991].

The split-and-nested-transaction model is an example of an *open-nested* model in which some component transactions (subtransactions) may decide to commit their effects in the database unilaterally. In Section 3.3, we will synthesize an open-nested 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 this 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 those of root transactions. Suppose we want the semantics of a split transaction to be similar to those of its splitting transaction. Thus, when a root transaction splits, it should split into two root transaction, and when a subtransaction splits, it should split into two sibling subtransactions. In this way, a split of 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 a *nested-split-transaction* model. Such 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 3.2.3.2. *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 3.2.1.1

(26..29) Axiom 16..19 of Definition 3.2.2.1

(30) $\forall t,\ t = t_a \vee t \in Descendant(t_a) \forall ob\ \forall p,q\ Conflict(p_t[ob], q_{t_b}[ob]) \wedge (\exists r(r_{t_a}[ob] \in CanAccess_{t_b}(t_a))) \wedge p_{t_a}[ob] \in H \Rightarrow (p_t[ob] \rightarrow q_{t_b}[ob])$

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

(32) $\mathsf{Split}_{t_a}[t_b, CanAccess_{t_b}(t_a)] \in H \Rightarrow (Parent(t_a) \neq \phi \Rightarrow (t_b \mathscr{W}\mathscr{D} Parent(t_a)) \wedge (Parent(t_a) \mathscr{C}\mathscr{D} t_b))$

(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. Otherwise, the model would have required that the split subtransaction be considered an ancestor 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 nonleaf 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 $t'_a s$ 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 nonleaf 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 Chrysanthis and Ramamritham [1990].

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 the CanAccess set at the time of a split, two sibling transactions can cooperate effectively 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., effects of 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 only 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 Mohan et al. [1992].)

3.3 Open-Nested-Transaction Model

In an open-nested-transaction model, component transactions may decide to commit or abort unilaterally. This model is particularly suitable in structuring applications that need to access data stored in preexisting databases or data repositories managed by systems that do not support any global commit protocol such as two-phase commit protocol. Example of such applications are telecommunication services and Computer-Integrated Manufacturing (CIM).

In this article, 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, $t_1, \ldots, t_n$. 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) and conflict set of the transactions of the type and the semantics of the events associated with a particular transaction.

3.3.1 *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 intertransaction 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.

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

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.

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

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

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

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*.

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

Recall that $\mathscr{SCD}$ stands for strong commit dependency whereby if t_i commits, $comp\_t_i$ 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* $\mathscr{BCD}$ dependency on compensating transactions.

$$\forall t_i \; t_j \in Comp\_Trs \left(\left(Commit_{t_i} \rightarrow Commit_{t_j} \right) \Rightarrow (comp\_t_i \; \mathscr{BCD} \; comp\_t_j) \right)$$

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

$$(t_j \; \mathscr{BCD} \; t_i) : \left(Begin_{t_j} \in H \Rightarrow \left(Commit_{t_i} \rightarrow Begin_{t_j} \right) \right)$$

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.

$$\forall t, \; ob, \; p \; p_t[ob] \in View_{comp\_t_i} \Rightarrow \exists q \; q_{t_i}[ob] \in H_{ct}$$

Since we assume that all component transactions, including noncompensatable ones, can commit at any time, noncompensatable components should not be allowed to commit their effects on objects when they commit. There are a number of ways to structure noncompensatable component transactions [Chrysanthis 1991; Chrysanthis and Ramamritham 1992]. 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.

$$\forall 0 \leq i \leq n$$
$$\left(t_i \notin Comp\_Trs \Rightarrow \left(\mathsf{Commit}_{t_i} \in H \Leftrightarrow Delegate_{t_i}[s, \; AccessSet_{t_i}] \in H \right) \right)$$

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 [Garcia-Molina and Salem 1987].

3.3.2 *Complete Specification.* Now let us put everything together. These axioms constitute the specifications of the open-nested-transaction model.

Definition 3.3.2.1. *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 noncompensatable 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 = {Begin, Commit, Abort}
(2) IE_t = {Begin}
(3) TE_t = {Commit, Abort}
(4) t satisfies the fundamental Axioms 1 to 4 (Definition 2.5.2.1)
(5) $View_{t_a} = H_{ct}$.
(6) $ConflictSet_t = \{p_{t'}[ob] \mid ResponsibleTr(p_{t'}[ob]) \neq t,\ Inprogress(p_{t'}[ob])\}$
(7) $\forall ob\ \exists p\ p_t[ob] \in H \Rightarrow (ob\ is\ atomic)$
(8) $\text{Commit}_t \in H \Rightarrow \neg(t\ \mathscr{C}_N^*\ t)$
(9) $\exists ob\ \exists p\ \exists t'\ Commit_{t_p}[p_{t'}[ob]] \in H \Rightarrow \text{Commit}_{t_p} \in H$
(10) $\text{Commit}_{t_p} \in H \Rightarrow$
$\forall ob\ \forall p\ \forall t'(p_{t'}[ob] \in AccessSet_{t_p} \Rightarrow Commit_{t_p}[p_{t'}[ob]] \in H)$
(11) $\exists ob\ \exists p\ \exists t'\ Abort_t[p_{t'}[ob]] \in H \Rightarrow \text{Abort}_t \in H$
(12) $\text{Abort}_t \in H \Rightarrow \forall ob\ \forall p\ \forall t'(p_{t'}[ob] \in AccessSet_t \Rightarrow Abort_t[p_{t'}[ob]] \in H)$
(13) $\forall t',\ ob,\ p\ p_{t'}[ob] \in View_{comp\_t_i} \Rightarrow \exists q\ q_{t_c}[ob] \in H_{ct}$
(14) $\forall t \notin Comp\_Trs\ \text{Commit}_t \in H \Leftrightarrow Delegate_t[s,\ AccessSet_t] \in H$
(15) $Begin_t \in H \Rightarrow ((t\ \mathscr{W}\mathscr{D}\ s) \wedge$
$(t \in VitalTrs \Rightarrow (s\ \mathscr{A}\mathscr{D}\ t)) \wedge (t \notin VitalTrs \Rightarrow (s\ \mathscr{C}'\mathscr{D}\ t)))$.
(16) $Abort_s \in H \Rightarrow \forall i\ (t_i \in Comp\_Trs \Rightarrow (comp\_t_i\ \mathscr{S}\mathscr{C}\mathscr{D}\ t_i))$
(17) $\forall t_i\ t_j \in Comp\_Trs\ ((Commit_{t_i} \rightarrow Commit_{t_j}) \Rightarrow (comp\_t_i\ \mathscr{B}\mathscr{C}\mathscr{D}\ comp\_t_j))$

In summary, Axioms 1 to 12 are similar to the corresponding ones of atomic transactions. All 12 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 must be acyclic, i.e., the transactions must be serializable taking into consideration the process of delegation (Axiom 8).

Axioms 9 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 noncompensatable transactions. When a noncompensatable component commits, it delegates its access set to its top-level transaction (Axiom 14). If a 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 [Garcia-Molina and Salem 1987]. 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 nonvital and noncompensatable.) This is still the case even if we consider the special case of an open-nested transaction all of whose component transactions are nonvital and noncompensatable 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.

We should point out that our open-nested model is representative of a class of open-nested models in the sense that it captures many of the common characteristics of the models in the class. The class includes s-transactions [Veijalaine and Eliassen 1992], sagas [Garcia-Molina and Salem 1987], poly-transactions [Sheth et al. 1992], DOM transactions [Buchmann et al. 1992], and Flex transactions [Bukhes et al. 1993].

3.4 Discussion

The exercise of synthesizing different transaction models 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 modify some of the properties to tailor a different transaction model. We can precisely delineate the differences be-

tween models and understand what contributes to the differences and similarities between transaction models.

Two related questions arise in the context of specifications: (1) are the specifications for a particular transaction model complete? (2) are the specifications consistent with the requirements of a particular model?

Just as it is difficult to show the "completeness" of a set of requirements specifications for a piece of software, it is difficult to show the completeness, in absolute terms, of a set of axioms pertaining to a model. This is because extended transactions can be endowed with "open-ended" semantics, whereby each transaction model can have model-specific significant events where the events have model-specific semantics. Let us consider an example. The chain transaction model is derived by adding one axiom to the specifications of the joint-transaction model. That is, the axioms defining the joint-transaction model are a subset of the axioms defining the chain transaction model. The extra axiom further constrains the occurrence of the *join* event and is motivated by the additional requirement associated with chain transactions, one that requires that only linear transaction sequences must be produced by the model. Thus, while the axioms for the joint-transaction model can be considered to be complete with respect to the requirements of the joint-transaction model, they are not complete with respect to the chain transaction model. However, the axioms of the chain transaction model (with the additional axiom) can be shown to satisfy the requirements of the chain transaction model and hence can be considered to be complete with respect to these requirements.

ACTA allows a modeler to specify both the high-level properties (requirements) of a model and the lower-level behavioral aspects of the model in terms of axioms. If the higher-level properties can be proven using the axioms then *with respect to the properties that have been proven*, the axioms can be considered to be complete. Also, then, the axiomatic specifications can be said to be consistent with the requirements of a particular model.

In some sense, it is also possible to talk about completeness of a set of axioms in absolute terms. Recall that a modeler is required to specify all the significant events that can occur and to provide the semantics of all the events. Also, he/she must specify the view and conflict set of all the different transaction types that can occur in a model. This will allow us to check whether all the necessary aspects of a transaction's effects on objects as well as on other transactions have been specified. However, since some of the semantics are model-specific, e.g., there are special constraints on the invocation of the join event in the chain transaction model, it is not possible to check whether all the semantics associated with an event have been specified, unless we check them *against* a set of higher-level requirements.

The analysis of whether a set of axioms satisfies a set of requirements can be carried out within ACTA, by using first-order logic-based proof methods. In the interest of space we included in an appendix the proofs for the properties of just one model, the nested-transaction model.

Finally, it is important to point out that ACTA is not restricted to a single version environment nor to a specific recovery or concurrency control scheme.

In ACTA versions can be captured by appropriately setting the View of a transaction.

—Each object ob in a multiversion environment needs to be annotated by its version number ob^1, $ob^2, \ldots, ob^n$. Then operations on a particular version can be included in the view of a transaction.

—Recall that $View_t = H_{et}$ implies that the view equals the current history and applies to in-place updates. $View_t = (H_{comm} \cup \forall p,\ ob \mid p_t[ob] \in H)$ states that the view equals the union of the committed history and the operations invoked by the transaction itself. This is the case when intention lists are used, i.e., deferred updates are done to the objects.

4. CONCLUSION

The ACTA transaction framework was motivated by a need to provide a RISC-like metamodel for treating extended transactions. What exactly makes up the ACTA framework? ACTA is a first-order logic-based formalism along with the precedence relation. Basically, ACTA allows a transaction modeler to specify the behavioral properties of transactions that adhere to a model in terms of (1) the set of events associated with a transaction model, (2) the semantics of these significant events, in terms of their effect on objects and other transactions, (3) the view of a transaction, and (4) the conflict set of a transaction. ACTA allows a modeler to specify also the high-level properties of a transaction model. One can verify then that these properties hold given the specifications for the model. The final point was not stressed much in this article because of our focus on synthesis. Specifically, we showed how the building blocks of ACTA serve as powerful tools for the development of new transaction models in a systematic and precise way. Thus, besides supporting the specification and analysis of existing transaction models [Chrysanthis 1991; Chrysanthis and Ramamritham 1990; 1991b], ACTA has the power to specify the requirements of new database applications.

New transaction models can be synthesized either by tailoring existing models or by starting from first principles. For instance, 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 cotransactions 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. Cotransactions are more flexible than reporting transactions in the sense that 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 the specifications of the components of the model.

ACTA has also been applied to derive from the original definition of the Saga model [Garcia-Molina and Salem 1987], 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 [Chrysanthis and Ramamritham 1992].

A variety of extended-transaction models, besides those referred to already in the article, have appeared in the literature. (See Elmagarmid [1992] for a description of some of these *extended transaction models*.) In Chrysanthis [1991], several of these models have been specified and analyzed. Other authors ([Buchmann et al. 1992], for example) have also used ACTA to specify the behaviors of their extended-transaction models.

Finally, even though we do not spell out the details in this article, the ACTA formalism can be used to show the correctness of a particular implementation of a transaction model by first formalizing the properties of the specific mechanisms used in the implementation and then showing that they will maintain the correctness properties of the model. In this context, it will be useful to investigate ways in which the ACTA primitives themselves can be used to drive the development of these mechanisms. This is in line with the work on the ConTract Model [Wächter and Reuter 1992] and CACS [Stemple and Morrison 1992] in which activities are made up of multiple (transaction-like) steps, with explicit dependency relationships specified between the steps. The system ensures that such dependencies hold when the steps execute.

APPENDIX. THE PROPERTIES OF NESTED TRANSACTIONS

Here, we state the recovery and concurrency properties of nested transactions and show how they follow from the axiomatic definition of nested transactions (Definition 3.2.1.1) developed in Section 3.2.1. Theorem A.5 which follows from all lemmas in this appendix captures the recovery properties whereas Theorem A.6 captures the concurrency properties of nested transactions.

LEMMA A.1. *Suppose t_c is a root or a child transaction in a nested-transaction structure. t_c is failure atomic.*

PROOF. For t_c to be failure atomic, t_c must satisfy the two conditions in the definition of failure atomicity (Definition 2.5.1.3). These can be shown by induction on the depth of the hierarchy where a root transaction is at depth 0.

Basis Step 1. Let t_c be at depth 0. That is, t_c is a root. Here, we first show that a root is *quasifailure-atomic* and then show that a root is also failure atomic.

(1) t_c satisfies the two conditions of quasifailure-atomicity (Definition 3.3):
   (a) Condition 1 (the "all" clause) follows directly from Axioms 12 and 13.
   (b) Condition 2 (the "none" clause) follows directly from Axioms 14 and 15.

(2) Axiom 20 does not apply to a root transaction since $Parent(t_c) = \phi$, and hence, a root does not delegate any of the operations in its $AccessSet_{t_c}$ when it commits. Consequently, $AccessSet_{t_c}$ contains all the operations invoked by t_c. Hence Axioms 12 to 15 also satisfy the two conditions in failure atomicity. Thus, t_c is failure atomic. More simply, since t_c has been shown to be quasifailure-atomic, since it does not delegate, it is also failure atomic.

Basis Step 2. Let t_c be a subtransaction at depth 1. That is, t_c is a child of the root.

(1) (Condition 1) If t_c commits, due to Axiom 20, all operations invoked by t_c are delegated to the root. Since the root is quasifailure-atomic, either all delegated operations will be aborted by the root or all will be committed by the root.

(2) (Condition 2) If an operation invoked by t_c aborts, due to Axioms 14 and 15 all operations invoked by t_c are aborted.

Thus, t_c satisfies the definition of failure atomicity.

Induction Step. Let us assume that subtransactions at depth $\leq k$ are failure atomic. Suppose t_c is at depth $k + 1$. Its parent, say t_p, must be at depth k.

(1) (Condition 1) If t_c commits, due to Axiom 20 all operations invoked by t_c are delegated to t_p. Since t_p is failure atomic (induction hypothesis) either all delegated operations will be aborted by t_p or one of its ancestors, or all will be committed by the root.

(2) (Condition 2) If an operation invoked by t_c aborts, due to axioms 14 and 15 all the operations invoked by t_c will be aborted.

Thus, t_c is failure atomic. □

LEMMA A.2 NO ORPHAN COMMITS. *Let H be a history of a nested transaction, t_p and t_c be transactions where t_p is the parent of t_c.*

$$\Big(\big((\mathsf{Commit}_{t_p} \in H) \wedge \neg(\mathsf{Commit}_{t_c} \rightarrow \mathsf{Commit}_{t_p})\big) \vee \big((\mathsf{Abort}_{t_p} \in H) \wedge \neg(\mathsf{Commit}_{t_c} \rightarrow \mathsf{Abort}_{t_p})\big)\Big) \Rightarrow (\mathsf{Abort}_{t_c} \in H)$$

Informally, this states that an orphan, i.e., a child whose parent either commits or aborts before it terminates, will be aborted.

PROOF. This lemma is derived by rewriting $(t_p \mathscr{CD} t_c)$ and $(t_c \mathscr{WD} t_p)$ logically, induced by Axiom 19.

(1) $(t_p \mathscr{CD} t_c)$
$\Leftrightarrow ((\text{Commit}_{t_p} \in H) \Rightarrow ((\text{Commit}_{t_c} \in H) \Rightarrow (\text{Commit}_{t_c} \rightarrow \text{Commit}_{t_p})))$
$\Leftrightarrow (\neg(\text{Commit}_{t_p} \in H) \vee \neg(\text{Commit}_{t_c} \in H) \vee (\text{Commit}_{t_c} \rightarrow \text{Commit}_{t_p}))$
$\Leftrightarrow (\neg(\neg(\text{Commit}_{t_p} \in H) \vee (\text{Commit}_{t_c} \rightarrow \text{Commit}_{t_p})) \Rightarrow \neg(\text{Commit}_{t_c} \in H))$
(given that H is a complete history, t_c would have been terminated by invoking either Commit or Abort (Axiom 6). Only one of these termination events can occur in H (Axiom 7). Thus, $\neg(\text{Commit}_{t_c} \in H) \Leftrightarrow (Abort_{t_c} \in H)$.
$\Leftrightarrow (((\text{Commit}_{t_p} \in H) \wedge \neg(\text{Commit}_{t_c} \rightarrow \text{Commit}_{t_p})) \Rightarrow (\text{Abort}_{t_c} \in H))$

(2) $(t_c \mathscr{WD} t_p) \Leftrightarrow ((\text{Abort}_{t_p} \in H) \Rightarrow (\neg(\text{Commit}_{t_c} \rightarrow \text{Abort}_{t_p}) \Rightarrow (\text{Abort}_{t_c} \in H)))$
$\Leftrightarrow (((\text{Abort}_{t_p} \in H) \wedge \neg(\text{Commit}_{t_c} \Leftrightarrow \text{Abort}_{t_p})) \Rightarrow (\text{Abort}_{t_c} \in H))$

(3) From (1) and (2),
$(((\text{Commit}_{t_p} \in H) \wedge \neg(\text{Commit}_{t_c} \rightarrow \text{Commit}_{t_p})) \vee$
$((\text{Abort}_{t_p} \in H) \wedge \neg(\text{Commit}_{t_c} \rightarrow \text{Abort}_{t_p}))) \Rightarrow (\text{Abort}_{t_c} \in H)$. □

LEMMA A.3. *If a root transaction t_0 aborts, all operations performed by t_0 and its descendants abort.*

$$\text{Abort}_{t_0} \in H \Rightarrow \forall t\ (t = t_0 \vee t \in Descendants(t_0))$$

$$\forall ob\ \forall p\ (p_t[ob] \in H \Rightarrow Abort[p_t[ob]] \in H)$$

PROOF. This follows from the *no-orphan-commits* lemma and the failure atomicity property of the root transaction and subtransactions. □

LEMMA A.4. *Operations are committed only by root transactions*: $\forall ob\ \forall p\ \forall t\ Commit_{t_r}[p_t[ob]] \in H \Rightarrow Parent(t_r) = \phi$

PROOF. This follows from Axiom 12, $\forall ob\ \forall t\ Commit_{t_r}[p_t[ob]] \in H \Rightarrow \text{Commit}_{t_r} \in H \wedge Parent(t_p) = \phi$. □

THEOREM A.5. *Nested transactions have the following recovery properties*:

(1) $\forall t_c$, *t_c is a root or a child transaction in a nested-transaction structure*; *t_c is failure atomic.*

(2) *An orphan subtransaction, i.e., a child t_c whose parent t_p either commits or aborts before it terminates, will be aborted (No Orphan Commits lemma)*:

$$\Big(\big((\text{Commit}_{t_p} \in H) \wedge \neg(\text{Commit}_{t_c} \rightarrow \text{Commit}_{t_p})\big) \vee \big((\text{Abort}_{t_p} \in H) \wedge \neg(\text{Commit}_{t_c} \rightarrow \text{Abort}_{t_p})\big)\Big) \Rightarrow (\text{Abort}_{t_c} \in H).$$

(3) *If a root transaction t_0 aborts, all operations performed by t_0 and its descendants abort*:

$$\text{Abort}_{t_0} \in H \Rightarrow \forall t\ (t = t_0 \vee t \in Descendants(t_0))$$

$$\forall ob\ \forall p\ (q_t[ob] \in H \Rightarrow Abort[p_t[ob]] \in H)$$

(4) *Operations are committed only by root transactions*:

$$\forall ob\ \forall p\ \forall t\ Commit_{t_r}[p_t[ob]] \in H \Rightarrow Parent(t_r) = \phi.$$

In the nested-transaction model, given that subtransactions delegate their operations to their parent upon commitment, root transactions are responsible for all the operations invoked by their committed descendants; the committed nonroot transactions do not commit any operations. The history of events relating to committed nested transactions will appear as though only the root transactions invoked operations on objects and, given the following theorem, the operations were invoked serializably.

THEOREM A.6. *A history of committed root transactions is serializable.*

PROOF. Let H be a history of committed root transactions. Consider a root transaction t. Given Axioms 10 and 20, $\mathscr{C}$ relation, and consequently $\mathscr{C}_N$ relation (recall from Section 3, $(t_i \mathscr{C} t_j) \Rightarrow (t_i \mathscr{C}_N t_j)$) is established between t and any other transaction t' if t (or any of its committed descendants) has invoked operations conflicting with those invoked by t' (or any of its committed descendants). Thus, given Axiom 11, since H_{t_i} contains only committed transactions, H_{t_i} is serializable (Definition 3.2). □

REFERENCES

BADRINATH, B., AND RAMAMRITHAM, K 1992. Semantics-based concurrency control: Beyond commutativity *ACM Trans. Database Syst. 17,* 1 (Mar.)

BERNSTEIN, P. A , HADZILACOS, V., AND GOODMAN, N 1987. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading, Mass

BUCHMANN, A , ÖZSU, M T., HORMICK, M , GEORGOKOPOULOS, O., AND MANOLA, F. A. 1992. A transaction model for active distributed object systems. In *Database Transaction Models for Advanced Applications.* A. K. Elmagarmid, Ed. Morgan Kaufmann, San Mateo, Calif., 123–158.

BUKHRES, O. A , CHEN, J , DU, W., ELMAGARMID, A. K., AND PEZZOLI, R. 1993. Interbase: An execution environment for heterogeneous software systems. *IEEE Comput. 28,* 8 (Aug.), 57–69.

CHRYSANTHIS, P. K. 1991. ACTA, a framework for modeling and reasoning about extended transactions. Ph.D. thesis, Dept. of Computer and Information Science, Univ of Massachusetts, Amherst.

CHRYSANTHIS, P K., AND RAMAMRITHAM, K. 1992. ACTA: The SAGA continues. in *Database Transaction Models for Advanced Applications.* Morgan Kaufmann, San Mateo, Calif.

CHRYSANTHIS, P. K., AND RAMAMRITHAM, K. 1991a. A unifying framework for transactions in competitive and cooperative environments. *IEEE Bull. Office Knowl Eng. 4* (Feb.), 3–21.

CHRYSANTHIS, P. K., AND RAMAMRITHAM, K. 1991b. A formalism for extended transaction models. In *Proceedings of the 17th International Conference on Very Large Data Bases* VLDB Endowment.

CHRYSANTHIS, P. K., AND RAMAMRITHAM, K. 1990. ACTA: A framework for specifying and reasoning about transaction structure and behavior. In *Proceedings of the ACM SIGMOD International Conference on Management of Data.* ACM, New York, 194–203.

CHRYSANTHIS, P. K , RAGHURAM, S., AND RAMAMRITHAM, K. 1991 Extracting concurrency from objects: A methodology. In *Proceedings of the ACM SIGMOD International Conference on Management of Data.* ACM, New York, 108–117.

DU, W , AND ELMAGARMID, A. K. 1989. Quasi serializability: A correctness criterion for global concurrency control in InterBase. In *Proceedings of the 15th International Conference on Very Large Databases.* VLDB, Endowment, 347–355

ELMAGARMID, A., (ED.) 1992 *Database Transaction Models for Advanced Applications.* Morgan Kaufmann, San Mateo, Calif

FERNANDEZ, M., AND ZDONIK, S. 1989. Transaction groups: A model for controlling cooperative transactions. In *Proceedings of the Workshop on Persistent Object Systems: Their Design, Implementation and Use.* 128–138

GARCIA-MOLINA, H., AND SALEM, K. 1987. SAGAS. In *Proceedings of the ACM SIGMOD International Conference on Management of Data.* ACM, New York, 249–259.

HERLIHY, M. P., AND WEIHL, W. 1988. Hybrid concurrency control for abstract data types. In *Proceedings of the 7th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems.* ACM, New York, 201–210.

KORTH, H. F., AND SPEEGLE, G. 1988. Formal models of correctness without serializability. In *Proceedings of the ACM SIGMOD International Conference on Management of Data.* ACM, New York, 379–386.

MOHAN, C., HADERLE, D., LINDSAY, B., PIRAHESH, H., AND SCHWARZ, P. 1992. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst. 17,* 1, 94–162.

MOSS, J. E. B. 1981. Nested transactions: An approach to reliable distributed computing. Ph.D. thesis, MIT, Cambridge, Mass.

PU, C., KAISER, G., AND HUTCHINSON, N. 1988. Split-transactions for open-ended activities. In *Proceedings of the 14th International Conference on Very Large Data Bases.* VLDB Endowment, 26–37.

SCHWARZ, P. M., AND SPECTOR, A. Z. 1984. Synchronizing shared abstract data types. *ACM Trans. Comput. Syst. 2,* 3 (Aug.), 223–250.

SKARRA, A. 1991. Localized correctness specifications for cooperating transactions in an object-oriented database. *IEEE Bull. Office Knowl. Eng. 4,* 1 (Feb.), 79–106.

SHETH, A., RUSINKIEWICZ, M., AND KARABATIS, G. 1992. Polytransactions: A mechanism for management of interdependent data. In *Database Transaction Models for Advanced Applications.* Morgan Kaufmann, San Mateo, Calif.

STEMPLE, D. W., AND MORRISON, R. 1992. Specifying flexible concurrency control schemes: An abstract operational approach. In *Annual Australian Computer Science Conference. Australian Comput. Sci. Commun. 14,* 2, 873–892.

VEIJALAINE, J., AND ELIASSEN, F. 1992. The S-transaction model. In *Database Transaction Models for Advanced Applications.* Morgan Kaufmann, San Mateo, Calif.

WÄCHTER, H., AND REUTER, A. 1992. The ConTract Model. In *Database Transaction Models for Advanced Applications.* Morgan Kaufmann, San Mateo, Calif.

WEIHL, W. 1984. Specification and implementation of atomic data types. Ph.D. thesis, MIT, Cambridge, Mass.

WEIHL, W. 1988. Commutativity-based concurrency control for abstract data types. *IEEE Trans Comput. 37,* 12 (Dec.), 1488–1505.

Received May 1993; revised November 1993; accepted April 1994