ACTA: The SAGA Continues

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

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

Abstract

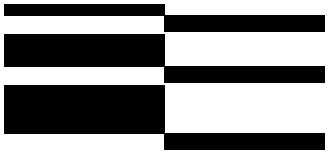
ACTA is a comprehensive transaction framework that permits a transaction modeler to specify the effects of *extended transactions* on each other and on objects in the database. ACTA allows the specification of (1) the interactions between transactions in terms of relationships between significant (transaction management) events, such as *begin, commit, abort, delegate, split*, and *join*, pertaining to different transactions and (2) transactions' effects on objects' state and concurrency status (i.e., synchronization state).

Various extended traditional models have been proposed to deal with applications that involve reactive (endless), open-ended (long-lived) and collaborative (interactive) activities. One such model is *Sagas* [GS87] A Saga is a set of relatively independent (component) transactions T_1, T_2,..., T_n which can interleave in any way with component transactions of other Sagas. Components can commit even before the Saga commits. However, if the Saga subsequently aborts, effects of the committed components are nullified through the invocation of compensating transactions.

After giving a brief introduction to the modeling primitives of ACTA, we illustrate their use by giving a complete formal characterization of Sagas. Subsequently, the reasoning power of ACTA is shown by proving properties of Sagas. Finally, the flexibility of ACTA is displayed through a series of variations to the original model of Sagas, each variation coming out of changes to the formal characterization of Sagas.

This is a reprint of Chapter 10 of *"Transactions Models for Advanced Database Applications,"* edited by A. K. Elmagarmid of Purdue University and published by Morgan Kaufmann (1992)

This material is based upon work supported by the N.S.F. under grant CDA-8922572.
E-mail: panos@cs.pitt.edu, krithi@cs.umass.edu.



Panos K. Chrysanthis
Krithi Ramamritham

10.1 Introduction

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

Although powerful, the transaction model [EGLT76, Gra81] found in traditional database systems is found lacking in *functionality* and *performance* when used for applications that involve reactive (endless), open-ended (long-lived) and collaborative (interactive) activities. Hence, various extensions to the traditional model have been proposed, referred to herein as *extended transactions* [Mos81, VRS86, BKK85, PKH88, KLS90, GGK+91, BHMC90, FZ89, SZ89, Elm91]. Compared to the traditional transaction model, these models associate "broader" interpretations with the four transaction notions mentioned above to provide enhanced functionality while increasing the potential for improved performance. Upon examining these *ad hoc* extensions to the traditional transaction model, one is prompted to seek answers to the following questions:

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

In attempting to answer these questions, we found a need for a common framework within which one can specify and reason about the nature of interactions between transactions in a particular model. This motivated the development of a comprehensive transaction framework, called *ACTA*[1], which characterizes the effects of transactions as per the taxonomy of Figure 10.1.

[1] ACTA means *actions* in Latin.

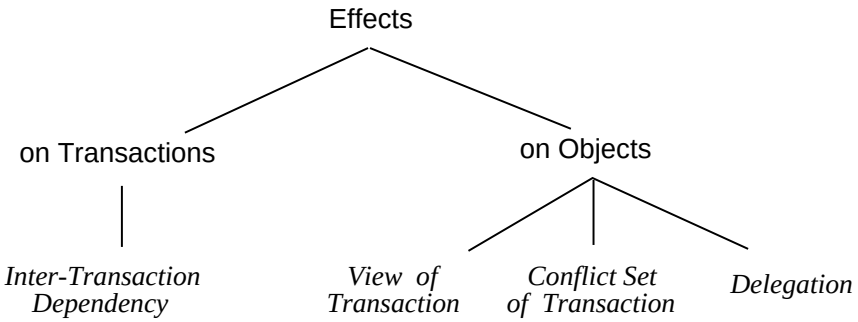


FIGURE 10.1
Dimensions of the ACTA Framework

After giving a brief introduction to the modeling primitives of ACTA, we illustrate its use by giving a complete formal characterization of *Sagas*, an extended transaction model developed to support long-lived activities. A Saga is a set of relatively independent (component) transactions which can interleave in any way with component transactions of other Sagas. Components can commit even before the Saga commits. However, if the Saga subsequently aborts, effects of the committed components are nullified through the invocation of compensating transactions.

The reasoning power of ACTA is shown by proving properties of Sagas. Finally, the flexibility of ACTA is displayed through a series of variations to the original model of Sagas, each variation coming out of changes to the formal characterization of Sagas. This will show the use of ACTA for "inventing" new transaction models by methodically varying the definition of existing models.

10.2 The Formal ACTA Framework

This section attempts to provide a concise yet complete introduction to ACTA. Subsection 10.2.1 provides some of the preliminary concepts underlying the ACTA formalism.

ACTA allows the specification of the effects of transactions on other transactions and also their effect on objects. Inter-transaction dependencies,

discussed in Subsection 10.2.2, form the basis for the former while visibility of and conflicts between operations on objects, discussed in Subsection 10.2.3 form the basis for the latter.

We would like to point out that with the evolution of ACTA, some aspects of ACTA introduced in earlier papers have changed. For example, the notion of *view set* introduced in [CR90] has been replaced in the current version by the notions of *view* and *conflict set* (see Subsection 10.2.3).

10.2.1 Preliminaries

Objects Events

A database is the entity that contains all the shared objects in a system. A transaction accesses and manipulates the objects in the database by invoking operations specific to individual objects. The *state* of an object is represented by its contents. Each object has a type, which defines a set of operations that provide the only means to create, change and examine the state of an object of that type. It is assumed that an operation always produces an output (return value), that is, it has an outcome (condition code) or a result. The result of an operation on an object depends on the state of the object. For a given state s of an object, we use $return(s, p)$ to denote the output produced by operation p, and $state(s, p)$ to denote the state produced after the execution of p.

DEFINITION

Invocation of an operation on an object is termed an object event. *The type of an object defines the operations and thus, the object events that pertain to it. We use $p_t[ob]$ to denote the object event corresponding to the invocation of the operation p on object ob by transaction t and OE_t to denote the set of object events that can be invoked[2] by transaction t (i.e., $p_t[ob] \in \mathrm{OE}_t$).*

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

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

DEFINITION

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.

DEFINITION

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

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

Commit and *Abort* operations are defined on every object for every operation. Invoked operations that have neither committed nor aborted are termed *in progress* operations. Typically, an operation is committed only if the invoking transaction commits and it is aborted only if the invoking transaction aborts. However, it is conceivable that an extended transaction may commit only a subset of its operations on an object while aborting the rest. Furthermore, through delegation (see Subsection 10.2.3), 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.

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 management primitives include Spawn, found in the nested transaction model, and Split and Join, found in the split transaction model [PKH88].

[3] As shown in Subsection 10.2.3, 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.

DEFINITION

Invocation of a transaction management primitive is termed a significant event. *A transaction model defines the significant events that pertain to transactions adhering to that model.* SE_t *denotes the set of significant events that is relevant to transaction* t.

ACTA does not *a priori* assume a given set of significant events nor does it associate any semantics with the significant events, but it 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

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

DEFINITION

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

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

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

Histories and Conditions on Event Occurrences

Fundamental to ACTA is the notion of *history* [BHG87] which represents the concurrent execution of a set of transactions T. ACTA captures the effects of transactions on other transactions and also their effects on objects through constraints on histories. This leads to definitions of transaction models in terms of a set of *axioms* which are either invariant assertions about the histories generated by the transactions adhering to the particular model

or explicit *preconditions* or *postconditions* of operations or transaction management primitives. Also, the correctness properties of different transaction models can be expressed in terms of the properties of the histories produced by these models.

DEFINITION

The execution of a transaction t is a partial order of events E_t with ordering relation $<_t$ where

1. $E_t \subseteq (OE_t \cup SE_t)$; and
2. $<_t$ denotes the temporal order in which the related events invoked by t occur.

In words, E_t contains events which are either object events allowed to be invoked by t or significant events related to t.

DEFINITION

A history *H of the concurrent execution of a set of transactions T contains all the events associated with the transactions in T and indicates the (partial) order in which these events occur. H_{ct} (the current history) is used to denote the history of events that occur until a point in time.*

The partial order of the operations in a history pertaining to T is consistent with the partial order $<_t$ of the events associated with each individual transaction t in T.

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

The predicate $\epsilon \rightarrow \epsilon'$ is true if event ϵ precedes event ϵ' in history H. It is false, otherwise. (Thus, $\epsilon \rightarrow \epsilon'$ implies that $\epsilon \in H$ and $\epsilon' \in H$.)

DEFINITION

$(\epsilon \in H) \Rightarrow Condition_H$, *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

$Condition_H \Rightarrow (\epsilon \in H)$ *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.

In specifying conditions over histories, we will find it useful to define the *projection* of a history H according to a given criterion $\wp$, denoted $Projection(H, \wp)$. For instance, $Projection(H, t)$, the projection of a history H on a specific transaction t yields the order of events related to t, denoted by H^t; whereas $Projection(H, ob)$, the projection of the history H on a specific object *ob* yields the history of operation invocations on the object, denoted by $H^{(ob)}$.

10.2.2 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.

DEFINITION

Dependency set, *denoted by DepSet, is a set of inter-transaction dependencies developed during the concurrent execution of a set of transactions* T. *Thus, DepSet is relative to a history* H. $DepSet_{ct}$ *(the current dependency set) is used to denote the set of dependencies until a point in time and hence, it is relative to* H_{ct}.

In the rest of this section, after formally specifying different types of dependencies, we identify the source of these dependencies.

Types of Dependencies

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

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

Strong-Commit Dependency $(t_j\ \mathcal{SCD}\ t_i)$: if transaction t_i commits then t_j commits; i.e., $(\mathsf{Commit}_{t_i} \in H) \Rightarrow (\mathsf{Commit}_{t_j} \in H)$.

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

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

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

Exclusion Dependency $(t_j\ \mathcal{ED}\ t_i)$: if t_i commits and t_j has begun executing, then t_j aborts (both t_i and t_j cannot commit); i.e.,
$(\mathsf{Commit}_{t_i} \in H) \Rightarrow ((\mathsf{Begin}_{t_j} \in H) \Rightarrow (\mathsf{Abort}_{t_j} \in H))$.

Force-Commit-on-Abort Dependency $(t_j\ \mathcal{CMD}\ t_i)$: if t_i aborts, t_j commits; i.e., $(\mathsf{Abort}_{t_i} \in H) \Rightarrow (\mathsf{Commit}_{t_j} \in H)$.

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

Serial Dependency $(t_j\ \mathcal{SD}\ t_i)$: transaction t_j cannot begin executing until t_i either commits or aborts; i.e.,
$(\mathsf{Begin}_{t_j} \in H) \Rightarrow (\epsilon \rightarrow \mathsf{Begin}_{t_j})$ where $\epsilon \in \{\mathsf{Commit}_{t_i}, \mathsf{Abort}_{t_i}\}$.

Begin-on-Commit Dependency $(t_j\ \mathcal{BCD}\ t_i)$: transaction t_j cannot begin executing until t_i commits; i.e., $(\mathsf{Begin}_{t_j} \in H) \Rightarrow (\mathsf{Commit}_{t_i} \rightarrow \mathsf{Begin}_{t_j})$.

Begin-on-Abort Dependency (t_j $\mathcal{BAD}$ t_i): transaction t_j cannot begin executing until t_i aborts; i.e., $(\mathsf{Begin}_{t_j} \in H) \Rightarrow (\mathsf{Abort}_{t_i} \rightarrow \mathsf{Begin}_{t_j})$.

Weak-begin-on-Commit Dependency (t_j $\mathcal{WCD}$ t_i): if t_i commits, t_j can begin executing after t_i commits; i.e.,
$(\mathsf{Begin}_{t_j} \in H) \Rightarrow ((\mathsf{Commit}_{t_i} \in H) \Rightarrow (\mathsf{Commit}_{t_i} \rightarrow \mathsf{Begin}_{t_j}))$.

The formal definitions of weak-abort dependency and abort dependency clearly reflect that weak-abort dependency is weaker than abort dependency. Weak-abort dependency is useful, for example, in specifying and reasoning about the properties of nested transactions [Mos81]. Begin-on-commit dependency, begin-on-abort dependency and force-commit-on-abort dependency are useful for compensating transactions [KLS90] and contingency transactions [BHMC90]. Begin-on-commit dependency and begin-on-abort dependency are special cases of serial dependency. The important difference between exclusion dependency and force-commit-on-abort dependency is that exclusion dependency allows both transactions to abort whereas force-commit-on-abort dependency does not.

We would like to note that this list of dependencies is *not* exhaustive. Other dependencies that involve significant events besides the Begin, Commit and Abort events, can be defined. As we will see in Section 10.5.4, when new significant events are associated with extended transactions, new dependencies may be specified in a similar manner. In this sense, ACTA is an open-ended framework.

Source of Dependencies

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

Dependencies due to Structure

The structure of an extended transaction defines its component transactions and the relationships between them. Dependencies can express these relationships and thus, can specify the links in the structure. For example, in hierarchically-structured nested transactions, the parent/child relationship is established at the time the child is *spawned*. This is expressed by a child transaction t_c establishing a weak-abort dependency on its parent t_p ($(t_c$ $\mathcal{WD}$ $t_p)$) and a parent establishing a commit dependency on its child ($(t_p$ $\mathcal{CD}$ $t_c)$).

Specifically, this is specified in terms of the postcondition of the Spawn event ($post(\mathsf{Spawn}_{t_p}[t_c])$):

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

The weak-abort dependency guarantees the abortion of an uncommitted child if its parent aborts. Note that this does not prevent the child from committing and making its effects on objects visible to its parent and siblings. (In nested transactions, when a child transaction commits, its effects are not made permanent in the database. They are just made visible to its parent. See [Chr91] for a precise formal definition of nested transactions.) The commit dependency of the parent on its child is preserved if (1) the parent does not commit before its child terminates, or (2) the child aborts in case its parent commits first. The weak-abort dependency together with the commit dependency says that an orphan, i.e., a child transaction whose parent has terminated, will not commit.

Other hierarchically-structured transactions may define various relationships between a parent and its child transactions. For example, in the transaction model proposed in [BHMC90, GGK+91] a parent can commit only if its *vital* children commit, i.e., a parent transaction has an abort dependency on its *vital* children t_v ($t_p \; \mathcal{AD} \; t_v$) (see Section 10.5.2). Child transactions may also have different dependencies with their parents if the transaction model supports various spawning or coupling modes [DHL90]. Sibling transactions may also be interrelated in several ways. For example, components of a *saga* [GS87] can be paired according to a compensated-for/compensating relationship [KLS90]. Relations between a compensated-for and compensating transactions as well as those between them and the saga can be specified via begin-on-commit dependency $\mathcal{BCD}$, begin-on-abort dependency $\mathcal{BAD}$, force-commit-on-abort dependency $\mathcal{CMD}$ and strong-commit dependency $\mathcal{SCD}$ (see Figure 10.5). In a similar fashion dependencies that occur in the presence of alternative transactions and contingency transactions [BHMC90] can also be specified (see Section 10.5.3).

Dependencies due to Behavior

Dependencies formed by the interactions of transactions over a shared object are determined by the object's synchronization properties. Broadly speaking, two operations conflict if the order of their execution matters. For example, in the traditional framework, a compatibility table [BHG87] of an

object *ob* expresses simple relations between conflicting operations. A conflict relation has the form

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

indicating that if transaction t_j invokes an operation p and later a transaction t_j invokes an operation q on the same object *ob*, then t_j should develop a dependency of type $\mathcal{D}$ on t_i. As we will see in the next section, ACTA allows conflict relations to be complex expressions involving different types of dependencies, operation arguments, and results, as well as operations on the same or different objects.

10.2.3 Objects and the Effects of Transactions on Objects

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

Conflicts between Operations and the Induced Dependencies

A history $H^{(ob)}$ of operation invocations on an object *ob*, $H^{(ob)} = p_1 \circ p_2 \circ ... \circ p_n$, indicates both the order of execution of the operations, (p_i precedes p_{i+1}), as well as the functional composition of operations. Thus, a state s of an object produced by a sequence of operations equals the state produced by applying the history $H^{(ob)}$ corresponding to the sequence of operations on the object's initial state s_0 ($s = state(s_0, H^{(ob)})$). For brevity, we will use $H^{(ob)}$ to denote the state of an object produced by $H^{(ob)}$, implicitly assuming initial state s_0.

DEFINITION

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

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

Two operations that do not conflict are compatible.

Thus, two operations conflict if their effects on the state of an object or their return values are not independent of their execution order. Since state changes are observed only via return values, the semantics of the return values can be considered in dealing with conflicting operations.

DEFINITION

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

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

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

In the rest of the paper, we will consider the situation when the effects are immediate. In this case, when an operation q follows operation p and q is return-value dependent on p, the transaction t_j invoking the operation q must abort q if for some reason the transaction t_i aborts p.

$$(return\text{-}value\text{-}dependent(p, q) \wedge (p_{t_i}[ob] \rightarrow q_{t_j}[ob])) \Rightarrow \\ ((Abort_{t_i}[p_{t_i}[ob]] \in H) \Rightarrow (Abort_{t_j}[q_{t_j}[ob]] \in H)).$$

This dependency ensures the correct behavior of objects in the presence of failure.

Motivated by this, in ACTA, the concurrency properties of an object are formally expressed in terms of *conflict relations* of the form:

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

where $Condition_H$ is typically a dependency relationship involving the transactions t_i and t_j invoking conflicting operations p and q on an object ob. Obviously, the absence of a conflict relation between two operations defined on an object indicates that the operations are compatible and do not induce any dependency[4].

This generality allows ACTA to encompass both object-specific and transaction-specific semantic information. First consider some object-specific semantics. *Commutativity* does not distinguish between return-value dependent and independent conflicts. It treats both the same and uses abort dependency for both:

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

Recoverability [BR90] avoids the unnecessary development of an abort dependency for return-value independent conflicts. Thus, recoverability induces the following conflict relations:

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

$$\textit{return-value-dependent}(p,q) \wedge (p_{t_i}[ob] \rightarrow q_{t_j}[ob]) \Rightarrow (t_j \ \mathcal{AD} \ t_i).$$

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

[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*.

if t_j commits before t_i commits, t_i must reread x in order to commit. This is captured by the following conflict relation:

$$(read_{t_i}[x] \rightarrow write_{t_j}[x]) \Rightarrow ((\mathsf{Commit}_{t_j} \rightarrow \mathsf{Commit}_{t_i}) \Rightarrow (\mathsf{Commit}_{t_j} \rightarrow read_{t_i}[x])).$$

This conflict relation cannot be derived solely from the object-specific semantics of the page. Clearly, transaction specific concurrency control might *not* achieve serializability but still preserves consistency.

In the example, t_i has to reread the page x when, subsequent to the first read, the page is written and committed by t_j. In general, t_i may need to invoke an operation on the same or a different object. For instance, instead of x, t_i may have to read a *scratch-pad* object which t_i and t_j use to determine and reconcile potential conflicts. Thus, ACTA allows the specification of operations that need to be controlled to produce correct histories as well as the specification of operations that *have* to occur in correct histories. These correspond to *conflicts* and *patterns* in [Ska91].

The $Condition_H$ in a conflict relation may include other significant events defined by the various transaction models. As an example, consider the significant event *Notify*, related to the notion of *notification* useful in a cooperative environment [FZ89]. For instance, the condition $Notify_{t_j}[(t_i \ \mathcal{CD} \ t_j)]$ will cause a commit dependency to be established from transaction t_i to t_j as well as *notify* t_j about the development of the commit dependency. Such compound conditions can be used to define a recoverability-based table in a cooperative environment. Transaction t_j can use the information about the existence of the commit dependency to postpone the invocation of another operation that causes a commit dependency of t_j on t_i, and thus postpone the formation of a circular commit dependency.

The generality of the conflict relations allows ACTA to capture different types of type-specific concurrency control discussed in the literature [SS84, HW88, Wei88, BR90, CRR91], and even to tailor them for cooperative environments.

Controlling Object Visibility

Visibility and Conflicts

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

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

$View_t$ is formally specified to be a subhistory derived by projecting events in H_{ct}:

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

In other words, the subhistory is constructed by eliminating any events in H_{ct} that do not satisfy the given $\mathcal{P}redicate$. $\mathcal{P}redicate$ depends on t, events in H_{ct} and inter-transaction dependencies $DepSet_{ct}$. For example, the view of a subtransaction t_c in the nested transaction model is defined to be the current history, i.e., $View_{t_c} = H_{ct}$, allowing t_c to view *the* most recent state of objects in the database.

For a more elaborate example, suppose that a subtransaction t_c is restricted to operate only on those objects that have been accessed by its parent t_p and is allowed to notice the changes done to them by its parent. The view of such a subtransaction t_c is defined as following form.

$$View_{t_c} = Projection\{H_{ct}, p_t[ob] | (t = t_c \vee t = t_p \vee (CommittedTr(t) \wedge \exists q \ (q_{t_p}[ob] \in H_{ct})))\}.$$

The predicate $CommittedTr(t)$ is true if transaction t has committed. Thus, t_c can see the changes done by committed transactions on the objects accessed by its parent.

DEFINITION

The conflict set *of a transaction t, denoted by $ConflictSet_t$, contains those in-progress operations with respect to which conflicts have to be determined.*

The composition of $ConflictSet_t$ is determined by the particular transaction model. It is specified via a predicate which can involve events invoked by t and any other transaction t_i, events in H_{ct}, and dependencies in $DepSet_{ct}$:

$$ConflictSet_t = \{p_{t_i}[ob] \mid \mathcal{P}redicate(t, t_i, H_{ct}, DepSet_{ct})\}.$$

A transaction t_j can invoke an operation on an object without conflicting with another transaction t_i if the operations in progress performed by t_i on the same object are in the view of t_j but are not included in the conflict set of t_j. Let us illustrate this by considering nested transactions. In nested transactions, a subtransaction t_c can access without conflicts any object currently accessed by one of its ancestors t_a. This is captured by

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

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

This states that any operation invoked by an ancestor of t_c is not contained in $ConflictSet_{t_c}$. For this reason, a transaction t_c can invoke an operation that conflicts with another in progress, invoked by its ancestor t_a, without forming a dependency.

At any given time, the current history H_{ct} and current dependency set $DepSet_{ct}$ exist. 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} and $DepSet_{ct}$ using the $View_t$ and $ConflictSet_t$. If its preconditions are satisfied, the new event is invoked and appended to the H_{ct} reflecting its occurrence.

The axiomatic definitions also specify how the dependency set is modified when a significant event is invoked. As we saw earlier, if an event is an object event, the operation semantics may also induce new dependencies to be added to $DepSet_{ct}$.

The degree of visibility allowed by a transaction model depends on the *width* of the views of the transactions in the model and on the *size* of their conflict sets. By width we mean the length of the subhistory specifying the view. A larger width makes more operations visible while a smaller size leads

to fewer conflicts permitting more operations to be performed without conflicting.

Delegation

DEFINITION

$ResponsibleTr(p_{t_i}[ob])$ 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

$Delegate_{t_i}[t_j, p_{t_i}[ob]]$ 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 $p_{t_i}[ob]$ in the DelegateSet.

Delegation has the following ramifications which are formally stated in [Chr91]:

- $ResponsibleTr(p_{t_i}[ob])$ is t_i, the event-invoker, unless t_i delegates $p_{t_i}[ob]$ to another transaction, say t_j, at which point it 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 suscript, 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 has no effect if the delegated operations have already been committed or aborted.
- Delegation redirects the dependencies induced by delegated operations from the delegator to the delegatee — dependencies are sort of responsibilities.

Note that delegation broadens the visibility of the delegatee and is useful in selectively making tentative or partial results as well as hints, such as, coordination information, accessible to other transactions.

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

DEFINITION

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

In nested transactions, when the root commits, its effects are made permanent in the database, whereas when a subtransaction commits, via inheritance, its effects are made visible to its parent transaction. The notion of inheritance used in nested transactions is an instance of delegation. Specifically, when a child transaction t_c commits, t_c delegates to its parent t_p all the operations that it is responsible for

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

Delegation need not occur only upon commit or abort but a transaction can delegate any of the operations in its access set to another transaction at any point during its execution. This is the case for Co-Transactions and Reporting Transactions described in [CR91, Chr91].

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 other's operations to be visible without producing conflicts, by delegating operations to each other, or by *notify*ing each other of their behavior. By being able to capture these aspects of transactions, the ACTA framework is applicable to cooperative environments.

10.3 Characterization of Atomic Transactions

Since the building blocks for a Saga are atomic transactions, we now give a formal characterization of atomic transactions.

Atomic transactions combine the properties of serializability and failure atomicity. These properties ensure that concurrent transactions execute without any interference as though they executed in some serial order, and that either all or none of a transaction's operations are performed.

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

DEFINITION

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

DEFINITION

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

DEFINITION

A set of transactions T is (conflict preserving) serializable *iff*
$\forall t \in T \; \neg(t \mathcal{C}^* t)$

DEFINITION

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

This definition implies that for an object to behave *correctly* it must ensure that when an operation aborts, any return-value dependent operation that follows it must also be aborted. It is not necessary for it to exhibit serial behavior, i.e., it is not necessary for the order in which the operations are executed by different transactions to be serializable. This definition ensures the correct behavior of objects in the presence of failures assuming immediate effects of operations on objects. Similarly, such dependencies can be defined for deferred effects.

DEFINITION

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

This definition states that the serializable behavior of an object is ensured by preventing transactions from forming cyclic $\mathcal{C}$ relationships.

DEFINITION

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

DEFINITION

Transaction t is failure atomic *if*

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

As mentioned earlier, failure atomicity implies that all or none of a transaction's operations are executed. In the above definition, the "all" clause is captured by condition 1 which states that if an operation invoked by a

transaction t is committed on an object, all the operations invoked by t are committed by t. 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 by t.

In the same way that serializability and failure atomicity were expressed above, other correctness properties of extended transactions such as *quasi serializability* [DE89], *predicatewise serializability* [KS88] and *quasi failure atomicity* [Chr91], can be expressed in ACTA [Chr91].

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

DEFINITION

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

I. $\forall \alpha \in \mathrm{IE}_t \ (\alpha \in H^t) \Rightarrow \nexists \beta \in \mathrm{IE}_t \ (\alpha \rightarrow \beta)$

II. $\forall \delta \in \mathrm{TE}_t \ \exists \alpha \in \mathrm{IE}_t \, (\delta \in H^t) \Rightarrow (\alpha \rightarrow \delta)$

III. $\forall \gamma \in \mathrm{TE}_t \ (\gamma \in H^t) \Rightarrow \nexists \delta \in \mathrm{TE}_t \ (\gamma \rightarrow \delta)$

IV. $\forall ob \forall p, (p_t[ob] \in H) \Rightarrow$
$((\exists \alpha \in \mathrm{IE}_t \, (\alpha \rightarrow p_t[ob])) \wedge (\exists \gamma \in \mathrm{TE}_t \, (p_t[ob] \rightarrow \gamma)))$

Axiom I prevents a transaction from being initiated by two different events. Axiom II states that if a transaction has terminated, it must have been previously initiated. Axiom III prevents a transaction from being terminated by two different termination events. The last axiom, Axiom IV, states that only in-progress transactions can invoke operations on objects.

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

DEFINITION

AXIOMATIC DEFINITION OF ATOMIC TRANSACTIONS
t denotes an atomic transaction.

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

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

Axiom 4 states that atomic transactions satisfy the fundamental axioms. With respect to the significant events of atomic transactions, the fundamental axioms mean the following:

1. the Begin event can be invoked at most once by a transaction
   $((\mathsf{Begin}_t \in H) \Rightarrow \neg(\mathsf{Begin}_t \rightarrow \mathsf{Begin}_t))$ [Axiom I],

2. only an initiated transaction can commit or abort
   $((\mathsf{Commit}_t \in H) \Rightarrow (\mathsf{Begin}_t \rightarrow \mathsf{Commit}_t)$, and
   $(\mathsf{Abort}_t \in H) \Rightarrow (\mathsf{Begin}_t \rightarrow \mathsf{Abort}_t))$ [Axiom II], and

3. an atomic transaction cannot be committed after it has been aborted
   $((\mathsf{Commit}_t \in H) \Rightarrow ((\mathsf{Abort}_t \notin H) \wedge \neg(\mathsf{Commit}_t \rightarrow \mathsf{Commit}_t)))$,
   and vice versa
   $((\mathsf{Abort}_t \in H) \Rightarrow ((\mathsf{Commit}_t \notin H) \wedge \neg(\mathsf{Abort}_t \rightarrow \mathsf{Abort}_t)))$ [Axiom III].

Axiom 5 specifies that a transaction sees the current state of the objects in the database. Axiom 6 states that conflicts have to be considered against all in-progress operations performed by different transactions. Axiom 7 specifies that all objects upon which an atomic transaction invokes an operation are atomic objects. That is, they detect conflicts and induce the appropriate dependencies. Axiom 8 states that an atomic transaction can commit only if it is not part of a cycle of $\mathcal{C}$ relations developed through the invocation of conflicting operations. Note that the atomicity property local to individual objects is not sufficient to guarantee serializable execution of concurrent transactions across all objects [Wei84]. Axiom 9 states that if an operation is committed on an object, the invoking transaction must commit, and Axiom 10 states that if a transaction commits, all the operations invoked by the transaction are committed.

Axioms 8, 9 and 10 define the semantics of the Commit event of atomic transactions in terms of the *Commit* operation defined on objects. Similarly, Axioms 11 and 12 define the semantics of the Abort event in terms of the *Abort* operation defined on objects. Axiom 11 states that if an operation is aborted on an object, the invoking transaction must abort, and Axiom 12 states that if a transaction aborts, all the operations invoked by the transaction are aborted.

Based on the above axioms, the failure atomicity and serializability properties of atomic transactions can be shown (see [Chr91]).

10.4 Characterization of Sagas

Sagas have been proposed as a transaction model for long lived activities. A saga is a set of relatively independent (component) transactions T_1, T_2,..., T_n which can interleave in any way with component transactions of other sagas. Component transactions within a saga execute in a predefined order which, in the simplest case, is either sequential or parallel (no order).

Each component transaction T_i $(0 \leq i < n)$ is associated with a compensating transaction CT_i. A compensating transaction CT_i undoes, from a semantic point of view, any effects of T_i, but does not necessarily restore the database to the state that existed when T_i began executing.

Both component and compensating transactions behave like atomic transactions in the sense that they have the ACID[5] properties. However, their

[5]ACID properties are Atomicity (or failure atomicity), Consistency, Isolation (or serializability) and Durability (or permanence).

behavior is constrained by certain dependencies. For example, a compensating transaction can commit only if its corresponding component transaction commits but the saga to which it belongs aborts.

Component transactions can commit without waiting for any other component transactions or the saga to commit. For this reason, sagas do not require a commit protocol as opposed, for example, to nested transactions. Due to their ACID properties, component transactions make their changes to objects effective in the database at their commitment times. Thus, isolation is limited to the component transaction level and sagas may view the partial results of other sagas. This means that each component transaction does not have to observe the same consistent database state produced by committed component transactions belonging to the same saga. Clearly, in sagas, consistency is not based on serializable executions.

A saga commits, i.e., successfully terminates, if all its component transactions commit in the prescribed order. Under sequential execution, the correct execution of a committed saga is:

$$T_1\ T_2 \cdots\ T_n$$

A saga is not failure atomic but neither can it execute partially. Thus, when a saga aborts, it has to compensate for the committed components by executing their corresponding compensating transactions. Compensating transactions are executed in the reverse order of commitment of the component transactions. Thus, in the sequential case, the correct execution of an aborted saga after the commitment of its k^{th} component transaction, T_k $(1 \leq k < n)$, is:

$$T_1\ T_2\ \cdots\ T_k\ CT_k\ CT_{k-1}\ \cdots\ CT_1$$

Note that the commitment of T_n implies the commitment of the whole saga and hence, T_n is not associated with a compensating transaction CT_n.

Now, let us express the basic properties of sagas with a set of axioms. Without loss of generality, let us focus on sagas whose components execute sequentially. As it will become clear below, the axiomatic definitions of sagas with different execution orders differ only in Axiom 18 which specifies the execution order. We will use $pre(e)$ and $post(e)$ to denote the preconditions and postconditions of an operation or a transaction management primitive e respectively.

DEFINITION

AXIOMATIC DEFINITION OF SAGAS

S denotes a saga with n component transactions.
T_i denotes a component transaction.
CT_i denotes a compensating transaction of T_i.
t denotes either a T_i or CT_i.

1. $SE_S = \{\text{Begin, Commit, Abort}\}$
2. $IE_S = \{\text{Begin}\}$
3. $TE_S = \{\text{Commit, Abort}\}$
4. $SE_t = \{\text{Begin, Commit, Abort}\}$
5. $IE_t = \{\text{Begin}\}$
6. $TE_t = \{\text{Commit, Abort}\}$
7. t satisfies the fundamental Axioms I to IV
8. $View_S = \phi$
9. $View_t = H_{ct}$
10. $ConflictSet_S = \phi$
11. $ConflictSet_t = \{p_{t'}[ob] \mid t' \neq t, \mathcal{I}nprogress(p_{t'}[ob])\}$
12. $\exists ob\ \exists p\ (Commit_t[p_t[ob]] \in H) \Rightarrow (\text{Commit}_t \in H)$
13. $(\text{Commit}_t \in H) \Rightarrow \forall ob\ \forall p\ ((p_t[ob] \in H) \Rightarrow (Commit_t[p_t[ob]] \in H))$
14. $\exists ob\ \exists p\ (Abort_t[p_t[ob]] \in H) \Rightarrow (\text{Abort}_t \in H)$
15. $(\text{Abort}_t \in H) \Rightarrow \forall ob\ \forall p\ ((p_t[ob] \in H) \Rightarrow (Abort_t[p_t[ob]] \in H))$
16. $\forall ob\ \exists p\ (p_t[ob] \in H) \Rightarrow (ob\ is\ atomic)$
17. $(\text{Commit}_t \in H) \Rightarrow \neg(t\mathcal{C}^*t)$
18. $post(\text{Begin}_S) \Rightarrow (((T_i\ \mathcal{BCD}\ T_{i-1}) \in DepSet_{ct}) \wedge$
    $((CT_j\ \mathcal{WCD}\ CT_{j+1}) \in DepSet_{ct}) \wedge$
    $((CT_{n-1}\ \mathcal{BAD}\ S) \in DepSet_{ct}))$
    where $1 < i \leq n$, and $1 \leq j < n-1$
19. $post(\text{Begin}_{T_i}) \Rightarrow (((S\ \mathcal{AD}\ T_i) \in DepSet_{ct}) \wedge$
    $((T_i\ \mathcal{WD}\ S) \in DepSet_{ct}) \wedge$
    $((CT_i\ \mathcal{BCD}\ T_i) \in DepSet_{ct}))$
    where $1 \leq i < n$
20. $post(\text{Commit}_{T_i}) \Rightarrow (((CT_i\ \mathcal{CMD}\ S) \in DepSet_{ct}) \wedge$
    $((CT_i\ \mathcal{BAD}\ S) \in DepSet_{ct}))$
    where $1 \leq i < n$
21. $post(\text{Begin}_{T_n}) \Rightarrow ((S\ \mathcal{SCD}\ T_n) \in DepSet_{ct})$

A transaction structure which conforms to a saga transaction model consists of three types of transactions, namely, *saga* transaction, *component*

transactions and *compensating transactions*. Axioms 1 and 4 state that each type of transaction is associated with the significant events Begin, Commit and Abort. Axioms 7, 9 and 11–17 capture the fact that component and compensating transactions have semantics similar to atomic transactions.

Saga transactions cannot directly operate on objects in the database [Axiom 8] — saga transactions may execute local operations that do not involve access to the database, e.g., test the outcome and return values of their component transactions[6] — this is the reason for the presence of a saga node, e.g., in Figure 10.3.

Axiom 18 specifies the execution order of the component transactions and their associated compensating transactions. A sequential (total) order is specified by establishing a begin-on-commit dependency $\mathcal{BCD}$ between every pair of component transactions. In a similar way, partial orders may be defined. Clearly, in the case of a parallel execution, Axiom 18 will be absent.

Axioms 19 specifies the relationship between a saga transaction and the component transactions. The composition relationship is captured by an abort dependency $\mathcal{AD}$ of the saga transaction on each of the component transactions and weak-abort dependencies $\mathcal{WD}$ of each component transaction on the saga transaction. This is induced at the time a component transaction begins its execution. The special relationship between a saga transaction and the last component T_n is captured by Axiom 21 in terms of a strong-commit dependency $\mathcal{SCD}$. A saga transaction's strong-commit dependency on T_n ensures that if T_n commits, the whole saga commits.

Axioms 19 and 20 pair component and compensating transactions according to a compensated-for/compensating relationship. This relationship is reflected by a begin-on-commit dependency $\mathcal{BCD}$ of the compensating transaction on its associated component transaction [Axiom 19] and a force-commit-on-abort dependency $\mathcal{CMD}$ and a begin-on-abort dependency $\mathcal{BAD}$ of the compensating transaction on the saga transaction [Axiom 20]. If a component transaction aborts and rolls back, there is no meaning for its compensating transaction to execute. On the other hand, if a component transaction commits, the compensating transaction gives the saga the ability to semantically undo its effects by inducing force-commit-on-abort and begin-on-abort dependencies between the compensating transaction and the saga transaction. The correct execution order of the compensating transactions is ensured by the weak-begin-on-commit dependency $\mathcal{WCD}$ between every pair of compensating transactions [Axiom 18]. The begin-on-abort dependency $\mathcal{BAD}$ of CT_{n-1} on

[6] Saga transactions may also handle the interface to the environment, e.g., users.

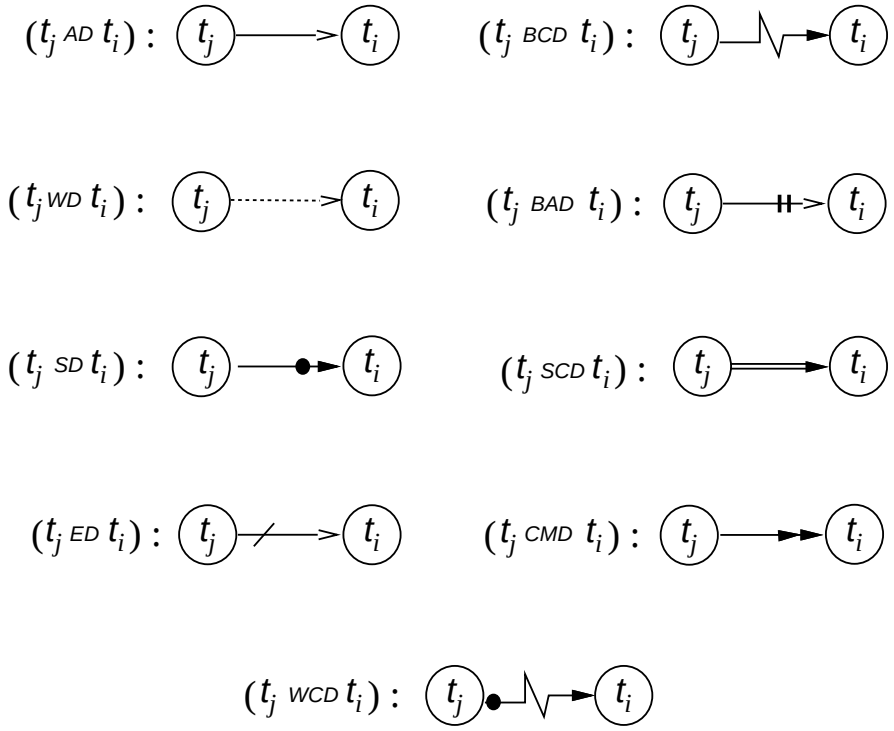


FIGURE 10.2
Dependencies relevant to Sagas

the saga transaction ensures that the compensating transactions do not execute prematurely and concurrently with the component transactions [Axiom 18]. By being the first on the chain of compensating transactions, CT_{n-1}'s outcome need to be considered first for the rest of the compensating transactions to execute.

Figure 10.2 shows the graph representation of the dependencies relevant to sagas. Figures 10.3–10.7 show five snapshots of the evolution of the structure of a saga (dynamics of intra-dependencies): (a) after a saga transaction has invoked begin, (b) when the first component transaction T_1 is in progress, (c) after T_1 commits, (d) when the second component T_2 is in progress and (e) when the last component T_n is in progress and consequently before the commitment of the saga. In these, a shaded node corresponds to a committed transaction.

This axiomatic definition captures the intended behavior of sagas. We now show some of the properties of the saga model using the axioms.

LEMMA 10.1 *Commitment of a Saga*

Let H be a history of a saga S with n component transactions.

$$(\mathsf{Commit}_S \in H) \Rightarrow \forall i, 1 < i \leq n\ (\mathsf{Commit}_{T_{i-1}} \rightarrow \mathsf{Commit}_{T_i})$$

Informally, this lemma states the history in which all component transactions commit in the required order.

PROOF

1. If S commits, T_i $(1 \leq i \leq n)$ must also have committed because of the abort dependency of S on T_i [Axiom 19] and the Fundamental Axiom III which states that a transaction has to either commit or abort [Axiom 7]:

   $$\forall i, 1 \leq i \leq n, ((\mathsf{Abort}_{T_i} \in H) \Rightarrow (\mathsf{Abort}_S \in H)) \Leftrightarrow ((\mathsf{Commit}_S \in H) \Rightarrow (\mathsf{Commit}_{T_i} \in H)).$$

2. Given T_i's $(1 < i \leq n)$ begin-on-commit dependency on T_{i-1} [Axiom 18]:

   $$\forall i, 1 < i \leq n\ ((\mathsf{Begin}_{T_i} \in H) \Rightarrow (\mathsf{Commit}_{T_{i-1}} \rightarrow \mathsf{Begin}_{T_i})),$$

   the Fundamental Axiom II:

   $$\forall i, 1 < i \leq n\ ((\mathsf{Commit}_{T_i} \in H) \Rightarrow (\mathsf{Begin}_{T_i} \rightarrow \mathsf{Commit}_{T_i})),$$

   and the semantics of the precedence relation,
   if T_i commits, then T_i commits after T_{i-1} commits:

   $$\forall i, 1 < i \leq n\ ((\mathsf{Commit}_{T_i} \in H) \Rightarrow (\mathsf{Commit}_{T_{i-1}} \rightarrow \mathsf{Commit}_{T_i}))$$

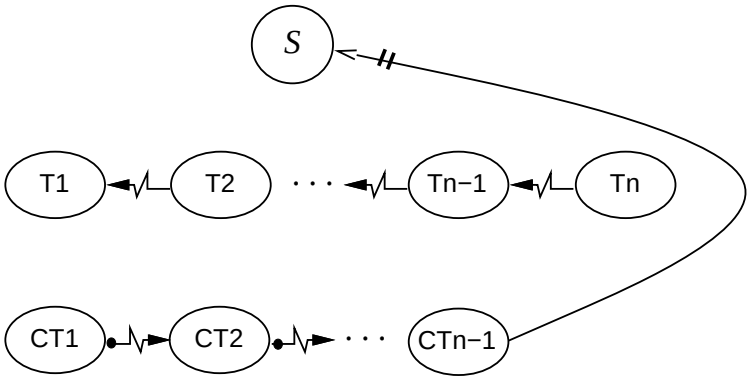


FIGURE 10.3
Structure of a just initiated Saga

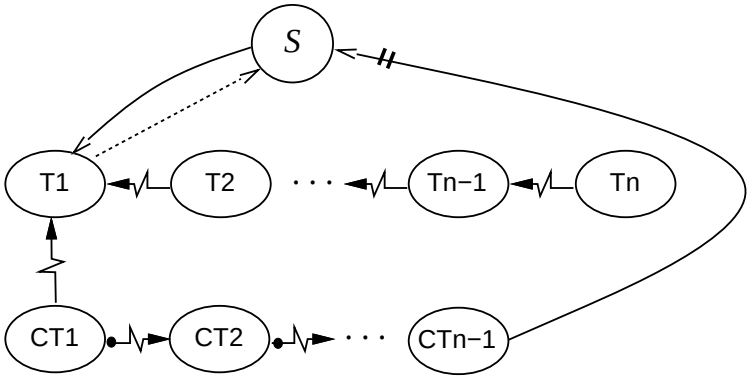


FIGURE 10.4
Structure of a Saga when component T_1 in progress

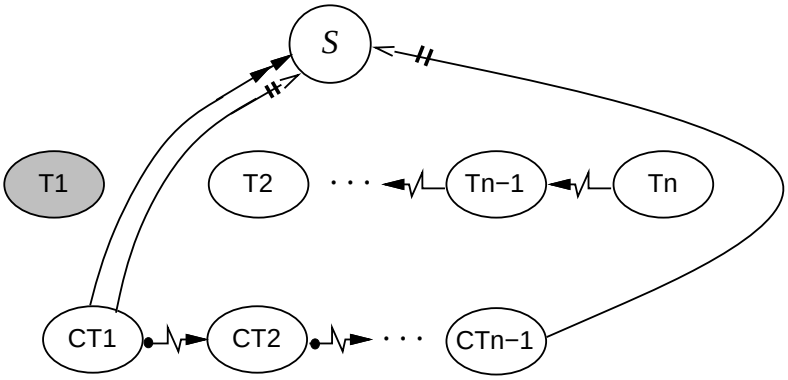


FIGURE 10.5
Structure of a Saga after component T_1 commits

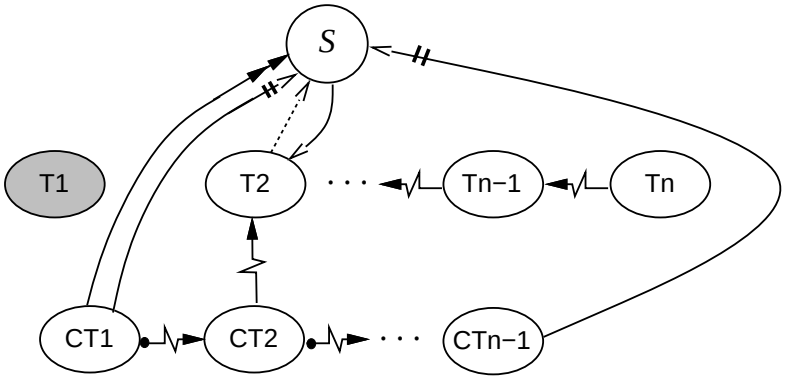


FIGURE 10.6
Structure of a Saga when component T_2 in progress

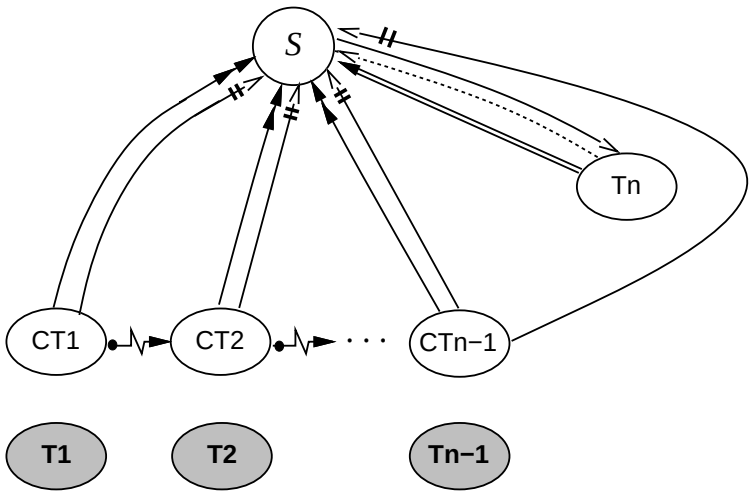


FIGURE 10.7
Structure of a Saga when component T_n in progress

3. Thus, from (1) and (2),

$$(\mathsf{Commit}_S \in H) \Rightarrow \forall i, 1 < i \leq n\ (\mathsf{Commit}_{T_{i-1}} \rightarrow \mathsf{Commit}_{T_i})$$

■

LEMMA 10.2 *Abortion of a Saga*

Let H be a history of a saga S with n component transactions.

$$\begin{aligned}
&(\mathsf{Abort}_S \in H) \Rightarrow \\
&\exists k, 1 \leq k \leq n\ \forall i, 1 < i < k-1 \\
&\quad (((\mathsf{Abort}_{t_k} \in H) \wedge (\mathsf{Commit}_{T_{i-1}} \rightarrow \mathsf{Commit}_{T_i}) \wedge \\
&\quad\ (\mathsf{Commit}_{T_{k-1}} \rightarrow \mathsf{Commit}_{CT_{k-1}}) \wedge (\mathsf{Commit}_{CT_i} \rightarrow \mathsf{Commit}_{CT_{i-1}})) \vee \\
&\exists k, 1 \leq k \leq n\ \forall i, 1 < i < k \\
&\quad ((\mathsf{Commit}_{T_{i-1}} \rightarrow \mathsf{Commit}_{T_i}) \wedge \\
&\quad\ (\mathsf{Commit}_{T_k} \rightarrow \mathsf{Commit}_{CT_k}) \wedge (\mathsf{Commit}_{CT_i} \rightarrow \mathsf{Commit}_{CT_{i-1}})))
\end{aligned}$$

Informally, this expresses the history in which for all committed components,

their compensating transactions commit in the required order. The first clause corresponds to the case in which a saga aborts while one of its components is in progress whereas the second clause corresponds to the case in which the saga aborts in between the execution of two of its components, i.e., after one of its components has committed and before the next one in order begins executing.

PROOF

Let us first consider the simple case of $k = 1$.

Case 1: If S aborts and T_1 has begun but not yet committed, T_1 is aborted due to the weak-abort dependency of T_1 on S [Axiom 19]. Since CT_1 has a begin-on-commit dependency on T_1 [Axiom 19], CT_1 never executes. This is the trivial case of an aborted saga:

$$(\mathsf{Abort}_S \in H) \Rightarrow (\mathsf{Abort}_{t_1} \in H)$$

Case 2:

1. If S aborts after T_1 commits and before T_2 begins, then CT_1 must commit due to the force-commit-on-abort dependency of CT_1 on S [Axiom 20]:
   $$(\mathsf{Abort}_S \in H) \Rightarrow (\mathsf{Commit}_{CT_1} \in H).$$
2. Given the begin-on-commit dependency of CT_1 on T_1, if CT_1 commits, then T_1 must have also committed (see Step 2 of lemma 10.1):
   $$(\mathsf{Commit}_{CT_1} \in H) \Rightarrow (\mathsf{Commit}_{T_1} \rightarrow \mathsf{Commit}_{CT_1})$$

Thus, from (1) and (2), $(\mathsf{Abort}_S \in H) \Rightarrow (\mathsf{Commit}_{T_1} \rightarrow \mathsf{Commit}_{CT_1})$.

Now let us consider the general case of $1 < k \leq n$.

Case 3:

3. If S aborts while T_k is in progress, T_k aborts, because of the weak-abort dependency of T_k on S. Consequently, CT_k is never initiated because of its begin-on-commit dependency on T_{k+1}:
   $(\mathsf{Abort}_S \in H) \Rightarrow (\mathsf{Abort}_{T_k} \in H))$.
   This also follows, if T_k aborts which causes S to abort due to its abort dependency on T_k.
4. If T_k is in progress, T_j $(1 \leq j < k)$ has committed in the specified order because of the begin-on-commit dependency between

the components:

$$\forall i, 1 < i \leq k-1\ (\mathsf{Commit}_{T_{i-1}} \rightarrow \mathsf{Commit}_{T_i}.$$

5. Given that T_j $(1 \leq j < k)$ have committed, CT_j has a force-commit-on-abort dependency on S. If S aborts, CT_j commits according to force-commit-on-abort dependency:

$$(\mathsf{Abort}_S \in H) \Rightarrow \forall j, 1 < j \leq k\ (\mathsf{Commit}_{CT_j} \in H).$$

6. Given the weak-begin-on-commit dependency of CT_j on CT_{j+1} [Axiom 18], if both CT_j and CT_{j+1} commit, CT_j commits after CT_{j+1} has committed (similar to (2)):

$$(\mathsf{Commit}_{CT_j} \in H) \Rightarrow (\mathsf{Commit}_{CT_{j+1}} \rightarrow \mathsf{Commit}_{CT_j}).$$

From (3), (4), (5) and (6),

$$(\mathsf{Abort}_S \in H) \Rightarrow \exists k, 1 \leq k \leq n\ \forall i, 1 < i < k-1$$
$$((\mathsf{Abort}_{t_k} \in H) \wedge (\mathsf{Commit}_{T_{i-1}} \rightarrow \mathsf{Commit}_{T_i}) \wedge$$
$$(\mathsf{Commit}_{T_{k-1}} \rightarrow \mathsf{Commit}_{CT_{k-1}}) \wedge (\mathsf{Commit}_{CT_i} \rightarrow \mathsf{Commit}_{CT_{i-1}}))$$

Case 4: The other general case in which S aborts after T_k commits and before T_{k+1} begins is similar to Case 3 without the step (3).

■

THEOREM 10.1

The component transactions of a saga produce one of the following committed histories:

1. $\forall i, 1 < i \leq n\ (\mathsf{Commit}_{T_{i-1}} \rightarrow \mathsf{Commit}_{T_i})$
2. $\exists k, 1 \leq k \leq n\ \forall i, 1 < i < k\ ((\mathsf{Commit}_{T_{i-1}} \rightarrow \mathsf{Commit}_{T_i}) \wedge$
   $(\mathsf{Commit}_{T_k} \rightarrow \mathsf{Commit}_{CT_k}) \wedge (\mathsf{Commit}_{CT_i} \rightarrow \mathsf{Commit}_{CT_{i-1}}))$

PROOF

This theorem follows from lemmas 10.1 and 10.2 and the committed projection of the history. ■

10.4.1 A Special Case of Sagas

A special case of sagas is a saga whose transaction structure does not have a saga transaction. The first component transaction T_1 marks the beginning of the saga as if it issues the Begin_S significant event, and the last transaction T_n commits the saga as if it issues the $Commit_S$ significant event.

Here is the axiomatic definition of the special Sagas.

DEFINITION

t denotes either a T_i, a component transaction, or a CT_i, a compensating transaction of T_i.

1. $SE_t = \{\mathsf{Begin}, \mathsf{Commit}, \mathsf{Abort}\}$
2. $IE_t = \{\mathsf{Begin}\}$
3. $TE_t = \{\mathsf{Commit}, \mathsf{Abort}\}$
4. t satisfies the fundamental Axioms I to IV
5. $View_t = H_{ct}$
6. $ConflictSet_t = \{p_{t'}[ob] \mid t' \neq t, \mathcal{I}nprogress(p_{t'}[ob])\}$
7. $\exists ob\ \exists p\ (Commit_t[p_t[ob]] \in H) \Rightarrow (\mathsf{Commit}_t \in H)$
8. $(\mathsf{Commit}_t \in H) \Rightarrow \forall ob\ \forall p\ ((p_t[ob] \in H) \Rightarrow (Commit_t[p_t[ob]] \in H))$
9. $\exists ob\ \exists p\ (Abort_t[p_t[ob]] \in H) \Rightarrow (\mathsf{Abort}_t \in H)$
10. $(\mathsf{Abort}_t \in H) \Rightarrow \forall ob\ \forall p\ ((p_t[ob] \in H) \Rightarrow (Abort_t[p_t[ob]] \in H))$
11. $\forall ob\ \exists p\ (p_t[ob] \in H) \Rightarrow (ob\ is\ atomic)$
12. $(\mathsf{Commit}_t \in H) \Rightarrow \neg(t\mathcal{C}^*t)$
13. $post(\mathsf{Begin}_{T_1}) \Rightarrow (((T_i\ \mathcal{BCD}\ T_{i-1}) \in DepSet_{ct}) \wedge$
    $((CT_j\ \mathcal{WCD}\ CT_{j+1}) \in DepSet_{ct})) \wedge$
    $((CT_{n-1}\ \mathcal{BAD}\ T_n) \in DepSet_{ct})$
    where $1 < i \leq n$ and $1 \leq j < n-1$
14. $post(\mathsf{Begin}_{T_i}) \Rightarrow ((CT_i\ \mathcal{BCD}\ T_i) \in DepSet_{ct})$, where $1 \leq i < n$
15. $post(\mathsf{Commit}_{T_i}) \Rightarrow (((CT_i\ \mathcal{CMD}\ T_{i+1}) \in DepSet_{ct}) \wedge$
    $((CT_i\ \mathcal{CMD}\ T_n) \in DepSet_{ct})$
    where $1 \leq i < n$

Beyond the obvious difference arising from discarding the axioms related to the saga transaction type — Axioms 1–3, 8 and 21, the substantive differences between this axiomatic definition and the original one are:

(a) The Begin_{T_1} event replaces the Begin_S event in all relevant axioms.

(b) Axiom 13 which replaces Axiom 18 of the original definition, substitutes S with T_n.

(c) Axiom 15 which corresponds to Axiom 20 of the original definition, induces an additional force-commit-on-abort dependency $\mathcal{CMD}$ of the compensating transaction CT_i on the component transaction T_{i+1} — the component that executes after CT_i's corresponding component transaction T_i.

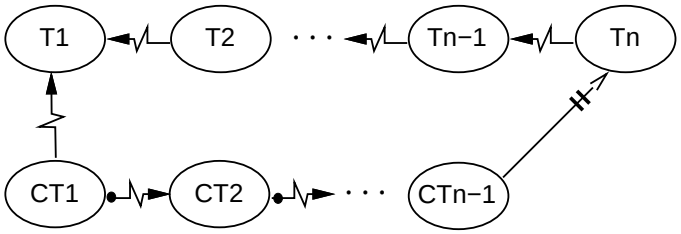


FIGURE 10.8
Structure of a special Saga before component T_1 commits

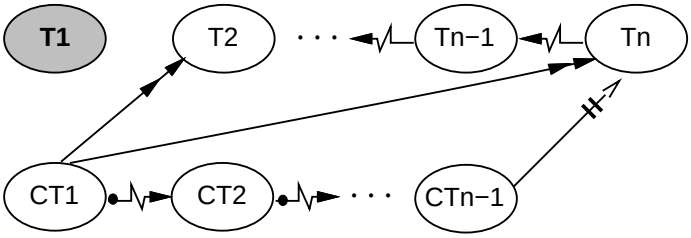


FIGURE 10.9
Structure of a special Saga after component T_1 commits

These differences reflect the fact that, in the special saga, T_1 and T_n carry the control role of the saga transaction, respectively, initiating and terminating the saga.

Figures 10.8 and 10.9 show snapshots of the structure of the special saga before and after the commitment of T_1.

Using these axioms, it is not hard to show that lemmas and theorem similar to lemmas 10.1 and 10.2 and theorem 10.1 hold for the special saga.

10.5 Variations of the Sagas Model

Since serializability and failure atomicity are not associated with a saga, a saga has no notion of commitment control beyond transaction boundaries. However, the commitment of a saga is dependent on the commitment of its components. A failure of a component forces the whole saga to abort. In this respect, sagas do not have the flexibility, e.g., of nested transactions, in being able to retry an aborted component, or to try an alternative component, or even to ignore a failed component.

In the following subsections, we show how sagas can be transformed to exhibit these properties by changing the dependencies defined in the original version of sagas. For the sake of brevity, we focus on the concepts and less on the formal aspects of the transformed sagas model. Where appropriate, we give the new (version of) axioms that formalize the properties of the new model.

10.5.1 Sagas with no Special Relation with Last Component

The original sagas call for a special relationship between a saga transaction and the last component transaction T_n because if T_n succeeds in committing then the saga commits as well. A saga thus lacks the flexibility of aborting after its last component has committed. Aborting a saga is easy and efficient as long as the information needed by the compensating transactions is available and easily accessible in the database. By committing a saga, this information is removed from the system.

To provide sagas with this flexibility, available, for example, in nested transactions, it is sufficient to treat the last component transaction as any other component. This means, first of all, that T_n needs to be associated with

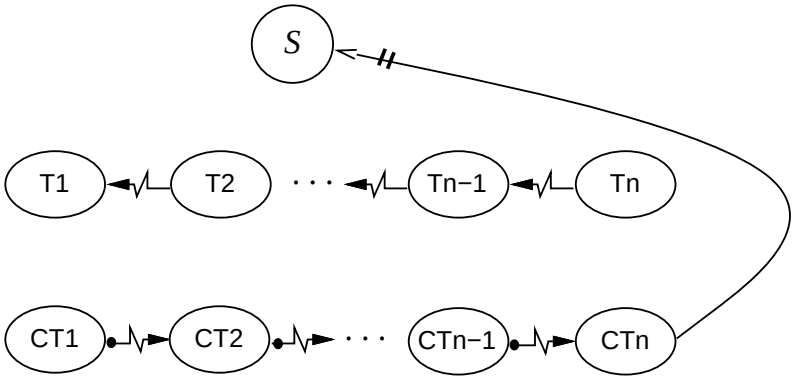


FIGURE 10.10
Structure of a Saga without special relation with T_n

a compensating transaction CT_n. The axiomatic definition of such a saga can be derived from the original axiomatic definition of sagas by dropping the last axiom, Axiom 21, and modifying Axioms 18 and 20 to include T_n. Figure 10.10 shows the structure of such a saga resulting from the modified Axiom 18. This corresponds to Figure 10.3 that represents the structure of the original saga.

10.5.2 Sagas with Vital Components

The relation between the saga and its component transactions is reflected, as stated by Axiom 19, by abort dependencies of the saga transaction on *each* of the component transactions and weak-abort dependencies of *each* of the component transactions on the saga.

Let us consider the case of a saga transaction that has *no* abort dependency on the first component transaction T_1 (see Figure 10.11). Since the abort dependencies of a saga transaction on the component transactions are the only constraints on the completion of a saga, T_1 can abort without preventing the saga from committing. In other words, the saga can ignore T_1 if it aborts (see Figure 10.12. Dotted nodes correspond to aborted transactions and transactions that cannot begin). This is not the case with the rest of the component transactions. Thus, the semantics of the relationship between a saga transaction and the component transactions changes with the removal

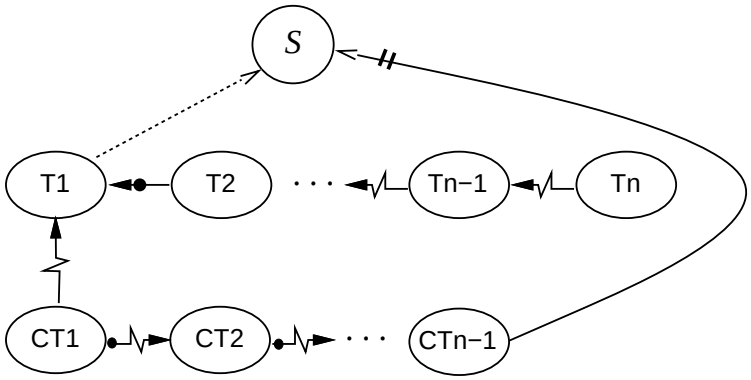


FIGURE 10.11
Structure of a Saga when non-vital component T_1 in progress

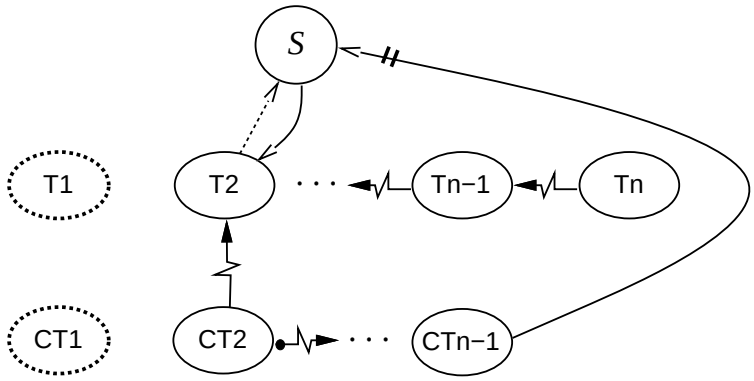


FIGURE 10.12
Structure of a Saga after non-vital component T_1 aborts

of the abort dependency. Specifically, there can be two types of relationships between sagas and its component transactions, namely, a *vital relation* and a *non-vital relation*. Consequently, component transactions can be distinguished as *vital* and *non-vital* transactions. A saga can commit only if its *vital* children commit. In the above case T_1 is not vital.

There is also a different relationship between vital and non-vital components which is captured by a serial dependency $\mathcal{SD}$ of a vital component on a non-vital component. The relationship between vital components remains the same as in the original saga captured by a begin-on-commit dependency.

The axiomatic definition of such a saga is the same as the original axiomatic definition except for Axioms 18 and 19 which need to be replaced by:

1. $VITAL_S = \{T_1\}$
2. $post(\mathsf{Begin}_S) \Rightarrow$
   $$(((T_{i-1} \in VITAL_S) \Rightarrow ((T_i \ \mathcal{BCD}\ T_{i-1}) \in DepSet_{ct})) \wedge$$
   $$(((T_{i-1} \notin VITAL_S) \Rightarrow ((T_i \ \mathcal{SD}\ T_{i-1}) \in DepSet_{ct})) \wedge$$
   $$((CT_j \ \mathcal{WCD}\ CT_{j+1}) \in DepSet_{ct}) \wedge$$
   $$((CT_{n-1} \ \mathcal{BAD}\ T_n) \in DepSet_{ct}))$$
   where $1 < i \leq n$, and $1 \leq j < n-1$
3. $post(Begin_{T_i}) \Rightarrow ((S \ \mathcal{AD}\ T_i) \in DepSet_{ct})$
   where $1 \leq i < n$ and $T_i \notin VITAL$
4. $post(Begin_{T_i}) \Rightarrow (((T_i \ \mathcal{WD}\ S) \in DepSet_{ct}) \wedge$
   $$((CT_i \ \mathcal{BCD}\ T_i) \in DepSet_{ct}))$$
   where $1 \leq i < n$

and corresponds to the axiomatic definition of the transaction model proposed in [BHMC90, GGK+91].

Using the above modified set of axioms, the correct committed histories of such a saga can be shown in a similar fashion as in Theorem 10.1.

THEOREM 10.2

The component transactions of a saga whose first component is a non-vital *transaction produce one of the following committed histories:*

1. *All component transactions commit in the required order:*
   $\forall i, 1 < i \leq n\ (\mathsf{Commit}_{T_{i-1}} \rightarrow \mathsf{Commit}_{T_i})$
2. *All vital component transactions commit in the required order:*
   $\forall i, 2 < i \leq n\ (\mathsf{Commit}_{T_{i-1}} \rightarrow \mathsf{Commit}_{T_i})$

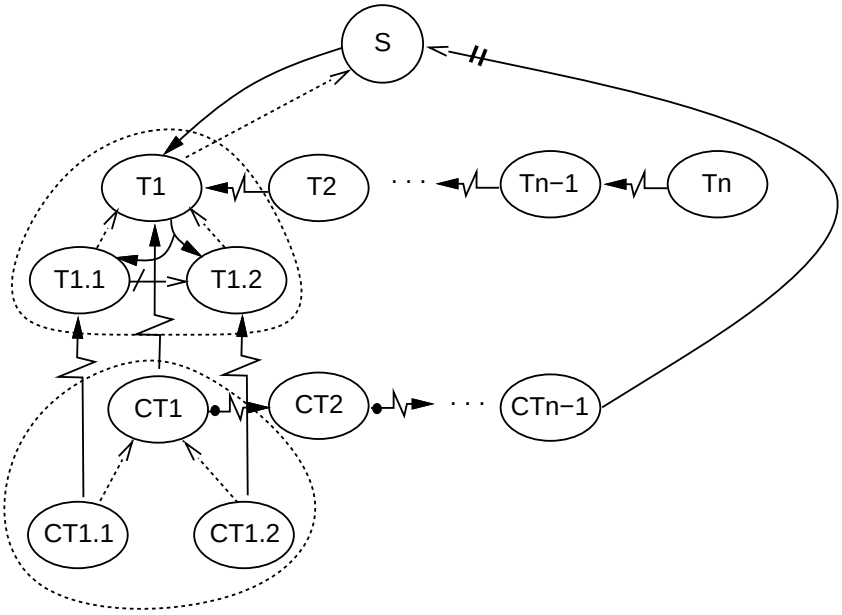


FIGURE 10.13
Saga structure when nested saga component T_1 in-progress

3. *For all committed components, their compensating transactions commit in the required order:*

$$\exists k, 1 \leq k \leq n \; \forall i, 1 < i < k$$
$$((\mathsf{Commit}_{T_{i-1}} \rightarrow \mathsf{Commit}_{T_i}) \wedge$$
$$(\mathsf{Commit}_{T_k} \rightarrow \mathsf{Commit}_{CT_k}) \wedge (\mathsf{Commit}_{CT_i} \rightarrow \mathsf{Commit}_{CT_{i-1}}))$$

10.5.3 Sagas of Sagas

The need for a more flexible transaction model created the concept of sagas. However, as we have already mentioned, sagas lack the flexibility to retry an aborted component, or to try an alternative component or even to ignore a failed component transaction. In the previous section, we saw that, by distinguishing between vital and non-vital transactions, a saga is able to

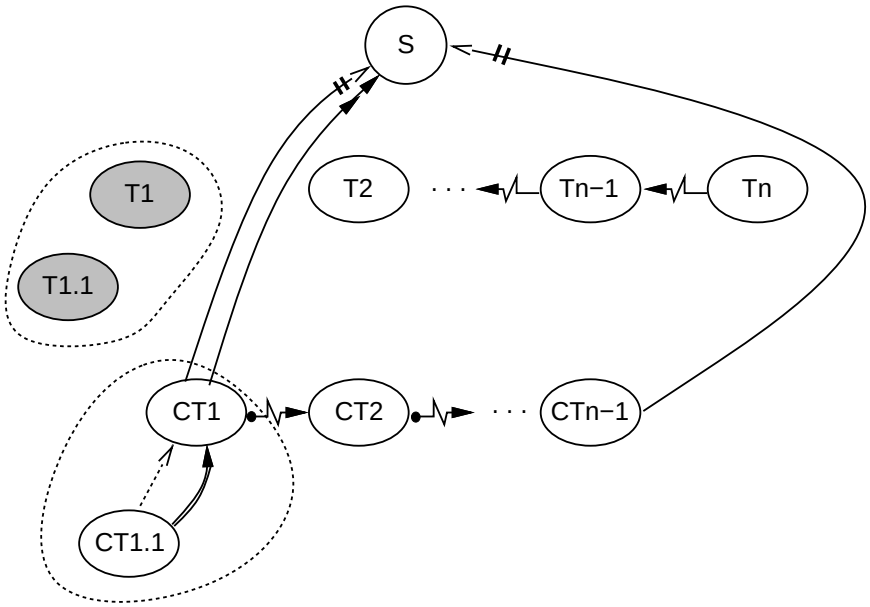


FIGURE 10.14
Saga structure after nested saga component T_1 commits

ignore a failed component. This was achieved fairly easily since the vitality of a component was manifested by the presence of an abort dependency of the saga transaction on the component transactions. Unfortunately, this is not sufficient when alternative transactions [BHMC90, ELLR90] and contingency transactions [BHMC90] are considered. For example, if two components exist where one is an alternative of the other, then both of them have to commit in order for the saga to commit. This contradicts the *at-most-one* semantics of alternative transactions — both alternatives cannot commit. This observation points to the concept of nested sagas[7] which are component transactions of sagas (Figures 10.13 and 10.14).

Dependencies between a nested saga transaction and the components of the nested saga are different from those of a (top) saga transaction on its

[7]Nested Sagas corresponds to a class of sagas with complex structure and hence, it is different from the nested saga model proposed in [GGK+91].

associated components. A nested saga is similar to a saga with non-vital components in the sense that a nested saga can commit even if some of its components abort. However, a nested saga has to abort if all of its component abort. This is captured through a *set-abort dependency* of the nested saga transaction on its associated component transactions:

Set-Abort Dependency $(t_j \ \mathcal{SAD} \ \{t_i | 1 \leq i \leq k\})$: if all t_i $(1 \leq i \leq k)$ abort then t_j aborts; i.e., $(\bigwedge_{1 \leq i \leq k}(\mathsf{Abort}_{t_i} \in H)) \Rightarrow (\mathsf{Abort}_{t_j} \in H)$.

In Figure 10.13, set-abort dependency corresponds to an arrow with multiple heads. Set-abort dependency brings out the fact that dependencies may involve more than two transactions.

Each component transaction of a nested saga has a weak-abort dependency $\mathcal{WD}$ on the nested saga transaction. As in the original saga, the weak-abort dependency ensures that if the nested saga aborts while its component transactions are still executing, its component transactions are also aborted.

A nested saga with this structure can exhibit different behaviors depending on the dependencies between its component transactions. In the simplest case, where no dependencies exist between the component transactions, nested sagas exhibit *at-least-one* semantics.

An *exclusion dependency* $\mathcal{ED}$ between the component transactions $T_{1.1}$ and $T_{1.2}$ of a nested saga T_1, as in Figure 10.13, captures the properties of alternative transactions. In particular, alternative transactions execute concurrently while the exclusion dependency ensures the *at-most-one* semantics.

Note that due to the semantics of the exclusion dependency $T_{1.1}$ cannot commit until $T_{1.2}$ aborts. This implies that $T_{1.2}$ is the preferable alternative. A second exclusion dependency from $T_{1.2}$ to $T_{1.1}$ will make both alternatives equally preferable.

Contingency transactions are a special case of alternative transactions in that they cannot execute concurrently. The sequential order of execution of contingency transactions is specified by means of begin-on-abort dependencies. Exclusion dependencies between the contingency transactions ensure the *at-most-one* semantics.

By being a component of a saga, a nested saga must be associated with a compensating transaction. In some special cases, a compensating transaction may be sufficient to compensate for any alternative or contingency transaction of a nested saga. It is often the case, however, that different transactions will need different compensating transactions. For this reason, a nested saga may be associated with a compensating saga whose components are the

compensating transactions of the component transactions of the nested saga. Begin-on-commit dependencies pair nested and compensating saga transactions and their associated component transactions (see Figure 10.13) reflecting their compensated-for/compensating relationship. If a component transaction aborts and rolls back, there is no meaning for its compensating transaction to execute. On the other hand, if a component transaction commits, a strong-commit dependency of a component transaction of a compensating saga on the compensating saga transaction propagates the effects of the force-commit-on-abort dependency of the compensating saga transaction on to the top saga transaction.

10.5.4 Sagas with Non-Compensatable Components

Sagas are built on the assumption that all their component transactions can be compensated for. There are many cases of component transaction that cannot be compensated for. There are even more cases of component transactions whose effects on objects can be compensated, but they involve real actions such as messages that cannot be semantically undone. In atomic transactions such actions are deferred until the commit time of the transaction. Since in sagas component transactions commit independently this approach is not directly applicable and hence sagas, as originally defined, are not applicable in such situations.

There are three different ways in which sagas can be extended to include non-compensatable component transactions. Each method is suitable for different situations and allows different levels of concurrency. The first method is applicable for sagas whose non-compensatable components execute concurrently. In such a situation, the weak-abort dependency of the component transaction on the saga transaction can be replaced with an abort-dependency coupling in this way the commitment of the saga with the commitment of the non-compensatable transactions. Thus, real actions are deferred until the saga commits.

Clearly, this method is not applicable for sequential executions because a non-compensatable component transaction T will block the execution of any component transaction which has a begin-on-commit dependency on T. In the second method, a new significant event, e.g., Finish, can be associated with non-compensatable transactions and a new dependency can be defined that relates the Begin and Finish events. (Recall that this is possible in ACTA

because ACTA is an open-ended framework allowing the introduction of new dependency relations.) Finish can be invoked by a transaction to terminate its access to shared objects in the database. However, Finish does not commit the operations invoked by a transaction on the shared objects. Thus, Finish does not replace Commit which is still needed to make the changes of a transaction effective in the database.

Defining **begin-on-finish dependency** (t_j $\mathcal{BFD}$ t_i) is straight forward: transaction t_j cannot begin execution until transaction t_i finishes; i.e.,

$(\mathsf{Begin}_{t_j} \in H) \Rightarrow (\mathsf{Finish}_{t_i} \rightarrow \mathsf{Begin}_{t_j})$.

Thus, in this second method, if a component transaction T_i is non-compensatable,

1. T_{i+1} $\mathcal{BFD}$ T_i,
2. $(\mathsf{Finish}_{T_i} \in H) \Rightarrow \nexists p \ \nexists ob \ (\mathsf{Finish}_{T_i} \rightarrow p_{T_i}[ob])$
3. T_i can invoke Commit only after invoking Finish:
   $(\mathsf{Commit}_{T_i} \in H) \Rightarrow (\mathsf{Finish}_{T_i} \rightarrow \mathsf{Commit}_{T_i})$,
4. if the saga aborts, T_i aborts after the components that execute following T_i have been compensated:
   $(\mathsf{Abort}_S \in H) \Rightarrow \forall j, i \leq j \leq n \ (\mathsf{Commit}_{CT_j} \rightarrow \mathsf{Abort}_{T_i})$, and
5. the saga commits iff T_i commits:
   $(\mathsf{Commit}_S \in H) \Leftrightarrow (\mathsf{Commit}_{T_i} \in H)$.

The third method does not require any additional significant events or any new dependencies. It simply structures non-compensatable transactions as subtransactions (*a lá* nested transactions) which at commit time delegate all the operations in their AccessSet, i.e., the operations that non-compensatable transactions have performed, to the saga.

Thus, in this last method, if T_i is a non-compensatable component of a saga S:

$$(\mathsf{Commit}_{T_i} \in H) \Leftrightarrow (Delegate_{T_i}[S, AccessSet_{T_i}] \in H).$$

If the saga aborts, all the effects of the operations in its Access Set are rolled back.

Subsection 10.5.1 through 10.5.4 discussed four different extensions to the original saga model and showed how their definitions are mutations of the original definition. Of course, it is possible to conceive of a model for sagas which combines two or more of these extensions. One can fairly easily develop the axiomatic definitions for such combined models given our discussions here. One such combination is a model similar to that of S–Transactions [VE91].

10.6 Conclusions

This paper shows how ACTA captures the (extended) functionality of a transaction model (1) by allowing the specification of significant events beyond commit and abort, (2) by allowing the specification of arbitrary transaction structures in terms of dependencies involving any significant event, (3) by supporting finer grain visibility for objects in the database by associating a view and a conflict set with each transaction and the notion of delegation, (4) and by facilitating object-specific and transaction-specific semantic-based concurrency control.

The application of ACTA to specify the properties of sagas revealed a number of possible variations to the saga model that are of practical interest. These involve (1) permitting a saga to commit even if a (non-vital) subset of the components of a saga aborted; (2) considering the compensatability of a saga component; (3) incorporating the notion of nested sagas within the saga model; and (4) combining alternative and contingency transaction model with the saga model. The ease with which it was possible to consider these variations, once the original Sagas model was characterized, speaks to the modeling capabilities of ACTA. Further, just as it was possible to show the correctness properties of original sagas model, it is possible to specify the correctness requirements of the extended Sagas and show the correctness given the characterizations of these extensions.

Bibliography

[BHG87] Bernstein, P. A., Hadzilacos, V., and Goodman, N. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading, MA, 1987.

[BHMC90] Buchmann, A., Hornick, M., Markatos, E., and Chronaki, C. Specification of a Transaction Mechanism for a Distributed Active Object System. In *Proceedings of the OOPSLA/ECOOP 90 Workshop on Transactions and Objects*, pages 1–9, 1990.

[BKK85] Bancilhon, F., Kim, W., and Korth, H. A model of CAD Transactions. In *Proceedings of the 11th International Conference on VLDB*, pages 25–33, 1985.

[BR90] Badrinath, B. and Ramamritham, K. Performance Evaluation of Semantics-based Multilevel Concurrency Control Protocols. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 163–172, 1990.

[Chr91] Chrysanthis, P. K. *ACTA, A Framework for Modeling and Reasoning about Extended Transactions*. PhD thesis, Department of Computer and Information Science, University of Massachusetts, Amherst, Massachusetts, 1991.

[CR90] Chrysanthis, P. K. and Ramamritham, K. ACTA: A Framework for Specifying and Reasoning about Transaction Structure and Behavior. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 194–203, 1990.

[CR91] Chrysanthis, P. K. and Ramamritham, K. A Unifying Framework for Transactions in Competitive and Cooperative Environments. *IEEE Bulletin on Office and Knowledge Engineering*, 4(1):3–21, 1991.

[CR91b] Chrysanthis, P. K. and Ramamritham, K. A Formalism for Extended Transaction Models. *Proceedings of the 17th International Conference on VLDB*, 1991.

[CRR91] Chrysanthis, P. K., Raghuram, S., and Ramamritham, K. Extracting Concurrency from Objects: A Methodology. In *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data*, 1991.

[DE89] Du, W. and Elmagarmid, A. K. Quasi Serializability: a Correctness Criterion for Global Concurrency Control in InterBase. In *Proceedings of the 15th International Conference on VLDB*, pages 347–355, 1989.

[DHL90] Dayal, U., Hsu, M., and Ladin, R. Organizing Long-Running Activities with Triggers and Transactions. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 204–214, 1990.

[EGLT76] Eswaran, K., Gray, J., Lorie, R., and Traiger, I. The Notion of Consistency and Predicate Locks in a Database System. *Communications of the ACM*, 19(11):624–633, 1976.

[ELLR90] Elmagarmid, A., Leu, Y., Litwin, W., and Rusinkiewicz, M. A Multidatabase Transaction Model for InterBase. In *Proceedings of the 16th International Conference on VLDB*, pages 507–518, 1990.

[Elm91] Elmagarmid A. (Issue Editor). Special Issue on Unconventional Transaction Management. *Bulletin of the IEEE Technical Committee on Data Engineering*, 14(1), 1991.

[FZ89] Fernandez, M. and Zdonik, S. Transaction Groups: A Model for Controlling Cooperative Transactions. In *Proceedings of the Workshop on Persistent Object Systems: Their Design, Implementation and Use*, pages 128–138, 1989.

[GGK+91] Garcia-Molina, H., Gawlick, D., Klein, J., Kleissner, K., and Salem, K. Modeling Long-Running Activities as Nested Sagas. *Bulletin of the IEEE Technical Committee on Data Engineering*, 14(1):14–18, 1991.

[Gra81] Gray, J. The Transaction Concept: Virtues and Limitations. In *Proceedings of the 7th International Conference on VLDB*, pages 144–154, 1981.

[GS87] Garcia-Molina, H. and Salem, K. SAGAS. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 249–259, 1987.

[HW88] Herlihy, M. P. and Weihl, W. Hybrid concurrency control for abstract data types. In *Proceedings of the 7th ACM symposium on Principles of Database Systems*, pages 201–210, 1988.

[KLS90] Korth, H. F., Levy, E., and Silberschatz, A. Compensating Transactions: A New Recovery Paradigm. In *Proceedings of the the 16th VLDB Conference*, pages 95–106, 1990.

[KS88] Korth, H. F. and Speegle, G. Formal Models of Correctness without Serializability. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 379–386, 1988.

[Mos81] Moss, J. E. B. *Nested Transactions: An approach to reliable distributed computing.* PhD thesis, Massachusetts Institute of Technology, 1981.

[PKH88] Pu, C., Kaiser, G., and Hutchinson, N. Split-Transactions for Open-Ended activities. In *Proceedings of the 14th International Conference on VLDB*, pages 26–37, 1988.

[Ska91] Skarra, A. Localized Correctness Specifications for Cooperating Transactions in an Object-Oriented Database. *IEEE Bulletin on Office and Knowledge Engineering*, 4(1):79–106, 1991.

[SS84] Schwarz, P. M. and Spector, A. Z. Synchronizing Shared Abstract Data Types. *ACM Transactions on Computer Systems*, 2(3):223–250, 1984.

[SZ89] Skarra, A. and Zdonik, S. Concurrency Control and Object-Oriented Databases. In *Object-Oriented Concepts, Databases, and Applications*, pages 395–421. ACM Press, 1989.

[VE91] Veijalaine, J. and Eliassen, F. The S–transaction Model. *Bulletin of the IEEE Technical Committee on Data Engineering*, 14(1):55–59, 1991.

[VRS86] Vinter, S., Ramamritham, K., and Stemple, D. Recoverable Actions in Gutenberg. In *Proceedings of the 6th International Conference on Distributed Computing Systems*, pages 242–249, 1986.

[Wei84] Weihl, W. *Specification and Implementation of Atomic Data Types*. PhD thesis, Massachusetts Institute of Technology, 545 Technology Square, Cambridge, MA, 1984.

[Wei88] Weihl, W. Commutativity-Based concurrency control for abstract data types. *IEEE Transactions on Computers*, 37(12):1488–1505, 1988.