ACTA, A Framework for Modeling and Reasoning About Extended Transactions

Panayiotis Kypros Chrysanthis
Department of Computer and Information Science
University of Massachusetts
Amherst, MA 01003

COINS Technical Report 91-90
September 1991

ACTA, A FRAMEWORK FOR MODELING AND REASONING ABOUT EXTENDED TRANSACTIONS

A Dissertation Presented

by

PANAYIOTIS KYPROS CHRYSANTHIS

Submitted to the Graduate School of the
University of Massachusetts in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

September 1991

Department of Computer and Information Science

To My Parents and My Wife

ACKNOWLEDGMENTS

It has been my privilege and good fortune to have Prof. Krithi Ramamritham as my advisor, mentor and friend. I will always be grateful to him for his directions, patience and support during my years as a graduate student. He inspires excellence in research and I have always admired his positive and open-minded style of criticism.

I will always be thankful to Prof. David Stemple for teaching me databases. Arguing with David was a challenge — a great way of sharpening my ideas. I will always remember his witty humor and rhetorical style. Comments from my other committee members, Profs. Jack Stankovic and C. Mani Krishna, have greatly benefited the dissertation. Jack also first inspired me to perform research in the area of distributed computing and gave me an opportunity to do so.

I would like to thank Prof. Alex Buchmann for his valuable critique of ACTA and for sharing with me his insights concerning careers in Computer Science. I have also received encouragement from Prof. Jim Kurose and Dr. Peter Bates.

In life, one makes very few good friends. Subhasish Mazumdar has been such a rare friend all these years at UMass. He has been a great source of encouragement in difficult times helping to clarify my thinking. It has been fun to discuss with both Ugo Buy and Victor Yodaiken; both have such a great humor. Douglas Niehaus, besides his invaluable help in proof-reading my drafts, made my move to Pittsburgh possible while the dissertation was in its crucial final stage. Badrinath, S. Raghuram and Chia Shen have been great colleagues. Without their support my task would have been much harder to accomplish.

I thank my officemates in the Distributed Systems Laboratory for making the long hours in the department pleasant. I greatly appreciate the assistance of the secretaries of the department and the staff of RCF and in particular of Renee Kumar, Betty Hardy, Barbara Gould, Valerie Caro and Lory Molesky.

Many thanks go to my friends outside the department who have made life enjoyable, especially Stella Kakavouli and Takis Metaxas. Last, but not least, I want to thank my friends Maria and Nikos Milonas and my godsons Theodoros and Constantinos for the happy moments, particularly during this last year.

ABSTRACT

ACTA, A FRAMEWORK FOR MODELING AND REASONING ABOUT EXTENDED TRANSACTIONS

SEPTEMBER 1991

PANAYIOTIS KYPROS CHRYSANTHIS

B.S., UNIVERSITY OF ATHENS, GREECE

M.S., UNIVERSITY OF MASSACHUSETTS

PH.D. UNIVERSITY OF MASSACHUSETTS

Directed by: Professor Krithivasan Ramamritham

The ability of *transactions* to mask the effects of concurrency and failures makes them appropriate building blocks for emerging advanced applications such as design environments. Several extensions have been proposed to the transaction model adopted in traditional database systems in order to support the *functional* and *performance* requirements of these complex systems. Often, however, given a particular extended transaction model, its properties as well as its scope of applicability are unclear.

To facilitate the formal description of transaction properties in an extended transaction model, we have developed ACTA, a comprehensive transaction framework. Specifically, using ACTA, one can specify and reason about the nature of interactions between extended transactions in a particular model. ACTA characterizes the semantics of interactions (1) in terms of different types of dependencies between transactions (e.g., commit dependency and abort dependency) and (2) in terms of transactions' effects on objects (their state and concurrency status, i.e., synchronization state). Through the former, one can specify relationships between significant (transaction management) events, such as *begin, commit, abort, delegate,*

split, and *join*, pertaining to different transactions. Also, conditions under which such events can occur can be specified precisely. Transactions' effects on object's state and status are specified by associating a *view* and a *conflict set* with each transaction and by stating how these are affected when significant events occur. A view of a transaction specifies the state of objects visible to that transaction while the transaction's conflict set contains those operations with respect to which conflicts need be considered.

The framework is capable of accommodating complex objects, as well as semantics-based multi-level concurrency control and recovery techniques. It has the potential to characterize transaction models that associate different semantics with the notions of *visibility*, *consistency*, *recovery*, and *permanence*. Its ability to capture the semantics of previously proposed transaction models is indicative of its generality. The reasoning capabilities of this framework have also been tested by using the framework to study the properties of new models derived either by combining existing transaction models or by extending existing transaction models using the ACTA formalism. In addition, ACTA can be used to show the correctness of a particular implementation of a transaction model by first formalizing the properties of the specific mechanisms used in the implementation and then showing that they will maintain the correctness properties of the model.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Motivation behind ACTA

With the advances in computing and communication technologies, applications of today and tomorrow go beyond typical commercial data processing. These new advanced applications are typically distributed and object based, i.e., designed in terms of an object-oriented paradigm. They include heterogeneous and federated databases, stock trading databases, multi-media databases, and databases used in computer aided design (CAD), computer aided software engineering (CASE), computer publishing, distributed operating systems, and distributed AI. The ability of *transactions* to mask the effects of concurrency and failures makes them appropriate building blocks for these complex systems.

Transaction properties are defined in terms of several important notions such as:

1. *Visibility*, referring to the ability of one transaction to see the results of another transaction *while* it is executing.

2. *Consistency*, referring to the correctness of the state of the database that a committed transaction produces.

3. *Recovery*, referring to the ability, in the event of failure, to take the database to some state that is considered correct.

4. *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.

Traditional transactions [40, 47] are based on the notion of *atomicity* and thus are often referred to as *atomic transactions*. Atomic transactions are based on a narrow interpretation of the above notions and provide failure atomicity, consistency, isolation and durability, i.e., the *ACIDity* properties. *Failure atomicity* means that either all or none of the transaction's operations are performed. *Consistency* means that a transaction maintains the integrity constraints on the database. *Isolation*, commonly known as *serializability*, means that concurrent transactions execute without any interference as though they were executed in some serial order. *Durability* means that all the changes made by a committed transaction become permanent in the database.

Although powerful, the transaction model adopted in traditional database systems is found lacking in *functionality* and *performance* when used for the new applications. Transactions in these applications tend to access many objects, involve lengthy computations, and are interactive, i.e., pause for input from the user. Even in those cases where activities with such characteristics can be modeled as traditional transactions, they degrade the effective concurrency of the system due to increased data contention, thus failing to meet the performance requirements. Furthermore, endless and collaborating activities which are often found in these systems, cannot be captured by traditional transactions due to serializability as the correctness requirement. Therefore, the need to capture reactive (endless), open-ended (long-lived) and collaborative (interactive) activities found in the new applications suggests the need for more flexible transaction models. Section 1.1.1 substantiates these points by giving an example from software design environments.

In order to fill this need for more flexible transaction models, various extensions to the traditional model have been proposed, referred to herein as *extended transactions*, which can support the needs of emerging applications [38, 39]. For example, Nested Transactions [73] have been proposed in the context of distributed languages to handle the problem of partial failures. Nested Transactions support only hierarchical computations similar to the ones that result from procedure calls. Recoverable Communicating Actions [97] which support arbitrary computation topologies, have

been proposed in the context of distributed operating systems where interactions are more complex. Cooperative Transactions [11], Split Transactions [82], Sagas [43, 45], Compensating Transactions [58], Contingency Transactions [21, 37], Multi-Coloured Actions [88] and Transaction Groups [41, 89] have also been suggested for capturing the interactions found in the new applications. 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 ask:

- What properties does a transaction model possess *vis a vis* visibility, consistency, recovery, and permanence?

If the properties can be precisely specified, it should be possible to ascertain the relation between a proposed model and another. For example,

- 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 aspects of one model be incorporated within another?

From an implementation viewpoint, the following question arises:

- What are the mechanisms needed for managing extended transactions? In particular, what are the mechanisms needed to achieve the desired correctness properties?

While it was tempting to develop more elaborate transaction models, thereby adding to the list of proposed models, we found the above questions even more intriguing. In attempting to answer these questions, we found a need for a common framework within which one can express the nature of interactions between extended transactions in a particular model. The idea was that given such a conceptual framework it would be possible to specify the effects of extended transactions and then reason about their properties.

1.1.1 An Application Requiring Extended Transactions

As mentioned earlier, traditional transactions are characterized by the ACIDity (failure atomicity, consistency, isolation and durability) properties. These are based on very narrow interpretations of the four transaction notions, namely, visibility, consistency, recovery, and permanence. More flexible interpretations which relax one or more of the ACIDity properties are necessary for dealing with transactions in new applications. As an example, consider a typical activity in a software development environment:

> A developer, modifying some of the procedures associated with an abstract data type module does the following: (a) For each procedure, (1) changes (the specification for the procedure within) the interface module, (2) changes the implementation of the procedure, and (3) changes the implementations of procedures that *use* the affected procedure; after all procedures have been handled, (b) creates a new object code for the module.

Here are some of the characteristics of this application:

- Modifying a module is an example of a long-duration transaction involving interactions with a user.

- Modifying a procedure involves modifying the interface specification of that procedure, modifying the implementation of that procedure, as well as other implementations that *use* this procedure, and creation of a new object code for the affected modules. Hence user-level operations have a multi-level structure, i.e., unlike reads and writes, they are not primitive operations.

- Suppose a user is making changes to multiple procedures within a single transaction in order to preserve consistency. If the system crashes, the developer does not want to lose all his changes. Instead, the developer would like all the modifications completed thus far to become permanent. In other words, failure of the transaction should not imply the failure of all the operations, and thus subtransactions should be the unit of recovery, i.e., failure atomicity.

- Recall that modifying a procedure may entail modifications to the implementations of other procedures that use this procedure. Hence it is quite plausible that two developers may attempt to modify the same procedure. Under serializability, the two requests must be executed one after another. This is likely to cause inordinate delays for the second developer since the second developer has to wait till the first commits. However, if one developer knows that another is also attempting to make modifications, then they can cooperate in order to optimize their modifications knowing each other's needs and changes made thus far. To accomplish this, when each developer changes a procedure, he/she maintains a set of notes in a temporary *scratch_pad* that the other developer may refer to. The resulting modifications may not be equivalent to either of the two possible serial executions of the modify requests but as long as the final implementation is consistent with the final state of the interface of the procedure, consistency of the module is maintained. Here we see a need for relaxing the serializability correctness requirement.

- The scratch_pad is an instance of an entity that does not have any permanence associated with it. It is discarded when the transaction that created it completes. But it may be used by transactions other than the one that created it and hence is not "local" to that transaction. Since they use each other's scratch_pads, there is a need for the two developers to synchronize their commitment since the scratch_pad that they created can not be deleted until both terminate.

This example illustrates a number of aspects of transactions in cooperative design environments. Partial changes made by one transaction may have to be made visible to another, while they are executing. The unit of consistency for modifying a procedure may involve changes to the implementation of that procedure and of other procedures that use it; the unit of recovery includes only the changes to a particular procedure. Thus, the unit of recovery or failure atomicity may be different from the unit of consistency. The scratch_pad is an instance of a shared entity that does not have any permanence associated with it. It is discarded when the transaction that created it completes. Finally, for efficiency purposes, serializability has to be

relaxed while still maintaining consistency. In summary, this example illustrates that visibility does not always have to be curtailed, consistency does not necessarily require serializability, recovery does not imply the complete restoration of the state when a system crash occurs, and permanence need not require all the results to be recorded in the database.

1.2 Extended Transactions: An Informal Definition

DEFINITION 1.1: An *extended transaction* consists of either a set of operations on objects that execute atomically in a predefined order, or a set of extended transactions with an explicitly given control related to the notions of visibility, consistency, recovery and permanence.

This recursive formulation implies that an extended transaction may exhibit a rich and complex internal structure. In contrast, traditional transactions have a flat single level structure. In this sense, the base case in this recursive definition of extended transactions is similar to a traditional transaction. A simple example of an extended transaction model is Nested Transactions [73].

Extended transactions are distinguishable from the *multi-level* transactions [75, 67, 8] first in that their internal structure is *explicit* and provided as a user facility, and second in that their component transactions are not necessarily *atomic*. Multi-level transactions have an *implicit* hierarchical internal structure which is a result of transactions invoking operations on complex objects. Thus, the operations are decomposable into sub-operations. Both operations and sub-operations are considered atomic. That is, for the user, a multi-level transaction is nothing but a set of atomic operations similar to a traditional transaction, and nesting is provided as a system facility.

The way that component transactions are combined to form extended transactions reflects the semantics of the applications. The semantics of an application may allow the definition of new weaker notions of conflicts among operations not possible with the information available only about objects and their types. For instance, operations invoked by two transactions can be interleaved as if they commuted, if the semantics of the application allow the dependencies between the transactions

to be ignored. This was illustrated by our example of two interacting developers. Clearly, transaction specific concurrency control might *not* achieve serializability but still preserves consistency. Also, based on an application's needs, in the event of failure of a transaction, changes made by completed *components* may be committed. Failed portions of a transaction can be retried, compensated, replaced by another (contingency) alternative, or even be ignored. These then are attractive for catering to the demanding functionality and performance needs of complex applications.

1.3 Contributions of This Work

ACTA is a comprehensive and flexible framework proposed to facilitate the formal description and reasoning of transaction properties in an extended transaction model. We chose the name *ACTA*, meaning *actions* in Latin, given the framework's appropriateness for expressing the properties of actions used to compose a computation.

In this dissertation, we have adopted the term *extended transactions*, preferring it over the other commonly used *unconventional transactions*. We believe it is more descriptive and also conforms with current literature. However, we do believe that a new term is needed for a computation whose actions possess "arbitrary" properties so as not to semantically overload the well defined concept of transactions. *Action-based computations* is a potential candidate.

The contributions of this work can be summarized as follows:

- The development of ACTA, a formal framework for specifying extended transactions.
  - ACTA is not *yet* another transaction model, but a model for transaction models.
  - ACTA allows for an intuitive, yet precise, definition of extended transactions by characterizing the semantics of interactions between extended transactions (1) in terms of different types of dependencies between transactions, and (2) in terms of transactions' effects on objects (their state and concurrency status, i.e., synchronization state).

- ACTA expresses the transaction properties of:
  - Visibility, in terms of the state of the objects visible to a transaction and the operations with respect to which conflicts need be considered.
  - Consistency, in terms of *allowable*, *required*, or *proscribed* relationships between significant events.
  - Recovery, in terms of the effects of significant events and *delegation*; the latter redefines the boundaries of failure relative to a set of objects.
  - Permanence, by separating operation commitment and transaction commitment.
- ACTA allows transaction inter-dependencies to relate significant events beyond the ubiquitous commit and abort events and thus, has the power to express interactions beyond those in atomic transactions. It is flexible enough to allow the definition of new dependencies as new significant events are associated with extended transactions.
- ACTA makes no assumptions about the structure of extended transactions and thus, is not restricted to hierarchically structured transactions.
- ACTA captures transaction inter-dependencies and the conflict relationships defined between operations on objects in a uniform manner; this simplifies reasoning about the interactions of transactions over shared objects and facilitates the use of transaction-specific semantics in determining conflicts between operations.

- The formalism
  - allows for the analysis of the correctness properties of extended transactions.
  - unifies existing transaction models making it possible to ascertain the relations between them, e.g., the similarities and differences between two transaction models.
  - facilitates the determination of whether two or more transaction models can be used in conjunction.

- supports the development and analysis of new extended transaction models in a systematic manner. New transaction models can be derived by combining and/or extending the specifications of existing transaction models.
- finally, even though we do not spell out the details in this thesis, the formalism can be used to show the correctness of a particular implementation of a transaction model by first formalizing the properties of the specific mechanisms used in the implementation and then showing that they will maintain the correctness properties of the model.

It is our view that the formal ACTA framework forms the basis for a *theory* of extended transactions.

1.4 Reading This Dissertation

This dissertation introduces the ACTA framework and examines its expressive power and reasoning capabilities. ACTA (Chapter 3) is based on First Order Predicate Calculus with a *precedence relation*. It is able to express both the transaction inter-dependencies and object conflict relations. It supports finer grain visibility of objects by means of two entities, namely, *view* and *conflict set*, associated with each transaction and the notion of *delegation*. The power of ACTA is studied along three dimensions: (1) its ability to express transaction-independent correctness criteria (Chapter 3), (2) its ability to express and reason about the properties of (extended) transactions that conform to a particular model (Chapter 4), and (3) its use to derive new extended transaction models (Chapter 5).

A chapter by chapter breakdown is:

Chapter1, the current chapter, introduces the concept of extended transaction models and given the proliferation of these models, argues about the need for a formal framework, such as ACTA, which can be used to specify and reason about extended transaction models.

Chapter 2 discusses related work in the areas of semantics-based concurrency control, concurrency control for complex objects, and non-serializable transaction

models. This shows how the existence of a framework like ACTA can be useful in unifying extant work aimed at providing for the newer database applications.

Chapter 3 introduces the formalism underlying ACTA and uses it to specify the correctness properties of failure atomicity, serializability, atomic objects, predicatewise serializability, cooperative serializability, quasi-serializability, setwise failure atomicity and quasi failure atomicity.

Chapter 4 focuses on the ability of ACTA to define existing transaction models. Specifically, it applies ACTA to model and reason about atomic transactions, distributed transactions, nested transactions, split and joint transactions, recoverable communicating actions, and sagas.

Chapter 5 examines the properties of new transaction models that are derived from existing ones using the ACTA formalism. Several such models are studied: First, chain transactions, reporting transactions and co-transactions are variations of joint transactions. The nested-split transaction model is a combination of the nested transaction and split transaction models. The rest are variations of Sagas which exhibit different degrees of flexibility.

Chapter 6 reviews the results presented in this dissertation, and summarizes the directions for future research.

CHAPTER 2

RELATED WORK

This chapter discusses related work in the areas of semantics-based concurrency control, concurrency control for complex objects, and non-serializable transaction models. This will show how the existence of a framework like ACTA can be useful in unifying extant work aimed at providing for the newer advanced applications.

2.1 Atomic Transactions

Traditional transactions have the properties of *(i) failure atomicity, (ii) consistency, (iii) isolation* and *(iv) durability*, i.e., ACIDity properties. *Failure atomicity* means that either all or none of the transaction's operations are performed. *Consistency* means that a transaction maintains the integrity constraints on the database. *Isolation*, commonly known as *serializability*, means that concurrent transactions execute without any interference as though they were executed in some serial order. *Durability* means that all the changes made by a committed transaction become permanent in the database.

In most schemes, serializability is based on the notion of conflicting operations and is called *conflict preserving serializability*. Two operations conflict when their effect is order-dependent. Conflict preserving serializability ensures that pairs of conflicting operations appear in the same order in two equivalent schedules [40, 48, 49]. Different versions of serializability such as *view* and *state* serializability, have different notions of effect [77, 78]. These different versions allow more concurrency than conflict preserving serializability but it is NP-complete to test whether a schedule is view or state serializable. Both pessimistic and optimistic schemes have been proposed to ensure serializability in centralized and distributed environments. Most

schemes are based on either two-phase locking or timestamp ordering [16]. There are also two general techniques to achieve failure atomicity [17]. One is based on the notion of careful replacement (commonly referred to as intentions lists) [96] and the other is based on maintaining execution logs [51].

To capture the needs of emerging information-intensive applications, several extensions have been made to the traditional data model. Instead of the read/write model of data, an abstract data type model has been advocated to capture the data in complex databases [50]. To support this, new locking mechanisms have been proposed [41, 61]. Multi-level transactions have been proposed to capture the semantics of transactions that access hierarchically structured complex objects. Since these extensions have a direct impact on our work, we provide a brief summary of these topics now.

2.1.1 Type-specific Concurrency Control

Several forms of type-specific concurrency control techniques based on the semantics of the operations defined on a type have been reported in the literature. *Commutativity* is the traditional semantic notion used to determine if two operations can be allowed to execute concurrently (e.g. two reads commute). In [100], commutativity is defined in terms of state machines as *forward commutativity*, assuming intentions lists based recovery, and *backward commutativity*, assuming log based recovery.

An alternative method for defining conflicts is based on *serial dependency relations* [53, 54]. *Invalidated-by* is such a serial-dependency relation: An operation p conflicts with another operation q if p can invalidate q by appearing earlier in a serial sequence. Specifically, if there exist operation sequences h_1 and h_2 such that $h_1 \cdot q \cdot h_2$ and $p \cdot h_1 \cdot h_2$ are legal sequences, but $p \cdot h_1 \cdot q \cdot h_2$ is not, then p *invalidates* q and q has a serial dependency on p. This criterion is feasible only if intentions lists based recovery is used. The return values of the operations and the parameters of the operations are used in the definition of conflicts. This definition is weaker than commutativity which requires equivalence of states. Serial dependency relations based concurrency control techniques, both optimistic and pessimistic, satisfy the

local atomicity property known as *hybrid atomicity* [99] which serializes transactions in commit order.

Recoverability is another criterion that is weaker than commutativity for defining conflicts [9, 10]. An operation q is recoverable relative to another operation p, if q returns the same value whether or not p is executed immediately before q. Transactions invoking operations p and q are required to commit in the order of invocation of the two operations. Since recoverability defines a (dynamically determined) order of commitment, it is stronger than serial dependency relations which postpone the commit order till the time of commitment of active transactions. When used with locking-based protocols, recoverability, like commutativity, avoids cascading aborts while also avoiding the delay in the processing of many non-commutative operations. It assumes a flexible recovery technique for handling the abortion of recoverable operations.

Compatibility of operations based on the formation of significant and insignificant dependencies between concurrent operations is described in [85]. For example, two concurrent read operations form an insignificant dependency and hence can be allowed to execute concurrently. The classification of dependencies as significant or insignificant is not uniform across types.

ACTA permits the incorporation of different type-specific concurrency control techniques in specifying the operation conflict relations for objects. Hence all of the techniques just discussed are of interest. More importantly, our work will take into account the fact that some of these schemes assume or dictate specific recovery or commit protocols. Hence their effect on transaction-specific concurrency and recovery protocols are of particular interest to us.

2.1.2 Concurrency Control for Complex Objects

The *multi-level transaction* model is suitable for building systems with hierarchically structured objects and layered systems in which each level is implemented on the abstraction of the next lower level. For each pair of adjacent levels, we can categorize the observable states of the higher level as *abstract* states and the states of the lower level which implement the abstract states, as *concrete* states. By drawing a distinction between *abstract* and *concrete* states, the notion of *abstract serializability*

is defined in terms of equivalence of abstract states. Similarly, *concrete serializability* is defined in terms of equivalence of concrete states. Concrete serializability corresponds to the traditional notion of serializability which requires equality of concrete states. Since many different concrete states in an implementation may represent the same abstract state, abstract serializability is less restrictive as a correctness criterion than concrete serializability. This means that abstractly serializable top level transactions may involve cyclic access to objects at lower levels. Abstract serializability can be obtained for top-level transactions by providing conflict-based serializability at each level.

The model of nested objects is another example of the multi-level model [67]. In this model, the nested scheduling protocol is bottom-up as opposed to the top-down protocol in [75] which first schedules higher level operations and then leaf level requests. The bottom-up protocol allows more concurrency than the top-down protocol by blocking operations at the lowest possible level. Recently, in [8], a new top-down protocol has been proposed which exhibits better performance than the one proposed in [75]. This new protocol schedules operations using an additional conflict criterion, called *relative conflict*. Two operations on an object have relative conflict if some of their suboperations conflict. Hence, by delaying such operations at each level, the possibility of deadlocks in the next lower level is eliminated.

A formal model for nested and multi-level computations is presented in [15]. Whereas this model is based on the serializability correctness criterion, ACTA is intended to capture correctness properties that go beyond serializability.

2.2 Spheres of Control

The notion of *spheres of control* proposed in [19, 31, 32] to structure data processing systems led to the simple notion of atomic transactions.

A *sphere of control* is a boundary around the effects of an arbitrary set of operations that can be unilaterally revoked or committed. Effects on objects are committed if no condition is attached to the use of the objects. A condition means that whoever sees the effects of operations on objects needs to "back out" if these effects are revoked. A dependency on the other hand is a conditional permission for

the use of objects. Whenever an operation is performed within a sphere of control, it develops a dependency on any other operation performed on the same object within a different sphere of control and whose effects on the object are uncommitted.

Spheres of control may be nested, maybe sequenced, and maybe parallelized. These interelationships among spheres of control may remain static or evolve dynamically. The semantics of applications define both the boundaries of a static sphere of control and the data flow relationships across the boundaries. The latter allows objects to be explicitly passed from a higher level sphere of control to a lower level, or objects to be attained within one sphere of control from another nested within it.

Spheres of control may be dynamically established as a result of dependencies induced by operations' interactions over shared objects. A dynamically created sphere of control is a higher level sphere of control around any two (lower) spheres where an operation within the first lower sphere has seen the uncommitted effects of operations within the second. The enclosing sphere of control defines the new boundaries for the commitment of the effects on the shared objects. That is, in the context of spheres of control, the commitment of the effects of operations within a sphere of control is not necessarily bound to the completion of the sphere.

The combination of localizing the effects of different operations and supporting object sharing beyond the boundaries of a sphere of control through dependencies constitutes a powerful tool for addressing recovery in an application independent fashion, and for maintaining consistency and integrity. ACTA is also founded on the concepts of localization and control of object sharing via dependencies; thus, it uniformly captures both static and dynamic structural properties through dependencies and expresses arbitrary data flow patterns across transaction boundaries.

2.3 Extended Transaction Models

In traditional transactions, interleaving of transactions is *implicitly* constrained by the concurrency properties of objects, i.e., operation conflicts. In an alternative approach, interleaving of transactions is *explicitly* constrained by the concurrency specifications of transactions. This approach may not achieve serializability but can

be used in such a way that consistency is preserved. Concurrency specifications are defined according to the semantics of the transactions and the data they manipulate.

In recent years, this approach has resulted in a proliferation of extended transaction models, e.g., [38, 39], since this approach provides the only means for dealing with the functionality and performance requirements of the new advanced applications (as illustrated by the example in Chapter 1). Here, we provide a brief overview of some well-known extended transaction models.

Maybe the best known extended transaction model is the Nested Transaction Model [73, 80] in which a transaction is composed of subtransactions that may execute concurrently. Subtransactions are serializable with respect to their siblings and failure atomic with respect to their parents. They can also abort independently without causing the abortion of the whole transaction. Nested Transactions have properties similar to multi-level transactions [15].

In the scheme proposed in [42], pairs of transactions are distinguished as being *compatible* or not. Compatible transactions, as with compatible operations, have semantic structure that allows them to execute concurrently. The scheme simplifies the specification of compatible transactions by classifying them into different semantic types.

The notion of compatible transactions has been generalized to several levels leading to the notion of *multi-level atomicity* [64]. In this model, transactions can belong to more than one semantic type. Each transaction type has different sets of breakpoints, inserted between the steps of a transaction at appropriate points. Steps of compatible transactions can be interleaved at these breakpoints.

In [64], the breakpoints are embedded (internal) in the body of the transactions whereas in [89] breakpoints are external to transactions, captured by a set of *patterns*. In this approach, *cooperative transactions* are grouped into *transaction groups*. A transaction group represents the unit of consistency and recovery. Components of a cooperative transaction may not produce consistent results. In fact, cooperative transactions do not have any of the properties of transactions and just represent different threads of controls. The consistency constraints for a transaction group are specified via the patterns. A pattern is a *state transition diagram* [12] which expresses goals and pieces of work to be done. Since goals in one pattern may

invalidate the goals of different patterns, *conflict specifications* among operations are needed to control the interleaving of concurrent patterns. Thus, the effects of a transaction group are considered to be consistent as long as all the steps of applicable patterns are executed (invoked exactly once by some cooperative transaction in the transaction group) and the patterns are interleaved correctly.

Just like transaction groups, *split transactions* [82] and *sagas* [45] have also been proposed to deal with the problem of long lived and cooperative transactions. In neither of these schemes, is there any clear notion of compatibility between transactions. A saga is like a two level nested transaction with traditional transactions as children transactions. Each child transaction is associated with an application-specific compensating transaction. A saga can be interleaved in any way with other sagas, but it cannot be partially executed. If a saga is interrupted, it either attempts to proceed by executing the missing transactions (i.e., forward recovery), or amends partial executions by invoking compensating transactions (i.e., backward recovery). Formal aspects of compensating transactions are discussed in [58, 62]. A variation of sagas [43] allows the characterization of children as *vital* or *non-vital* where the abortion of a *vital* child causes the abortion of the transaction. Sagas are appropriate in applications where each child transaction does not have to observe the same consistent database state.

Contingency transactions [21] are related to forward recovery, as opposed to compensating transactions which are related to backward recovery. A contingency transaction is invoked upon the failure of a transaction to accomplish a goal similar to that of the failed transaction.

In the context of long and cooperative transactions, the *Recoverable Communicating Actions (RCA)* model has been proposed to deal with the problem of non hierarchical computations [97]. In this model, an action, the *sender*, is allowed to communicate with another action, the *receiver*, by exchanging objects resulting in an *abort-dependency* of the receiver on the sender. If the sender aborts then the receiver must abort as a result of the dependency. However, partial failures are tolerated since an action may abort without aborting the action on which it has developed an abort-dependency.

Finally, the proposed notions of *compound transactions* [87] and *cooperative transactions* [57, 60] use the same correctness criterion, one which permits non-serializable executions while satisfying the individual postconditions of the transactions. In the case of compound transactions the criterion is known as *setwise serializability* whereas in the cooperative transactions case it is known as *predicatewise serializability*. Objects in both schemes are grouped into sets based on the database consistency constraints. In the compound transaction scheme, it is assumed that each set, called an *atomic data set*, has independent consistency constraints and that individual transactions operate only on a particular atomic data set. There is no such assumption in the case of cooperative transactions. Thus, it is possible for cooperative transactions to be serialized in different orders with regard to different atomic data sets.

While all of these extensions to the traditional transaction model appear to be reasonable and perhaps one could come up with more of them, the following is not always clear: Which aspects of a model take it beyond serializability? What correctness properties does a given transaction model satisfy? Where exactly do two transaction models differ? Is it reasonable to use one of the extended transaction models in conjunction with another?

These are some of the questions that prompted the work on ACTA.

2.4 Extensible Databases

A number of distributed systems use transactions operating on abstract data types. In these systems both locking techniques and recovery techniques for shared abstract data types are investigated. These include Argus [63, 99], Clouds [3, 4], ISIS [18], TABS [84, 92] and its successor Camelot [93].

Since each application has different requirements, the notion of *extensible databases* has been proposed which allows for the tailoring of the database system to the needs of a particular application. A number of projects are centered around the research and development of extensible databases. These include projects such as EXODUS [24, 25], PROBE [34, 66], POSTGRES [94], GemStone [30, 65], STARBURST [86], ORION [46, 56] and GENESIS [13, 14]. Issues such as data models, storage

structures, efficient access techniques and efficient transaction management are among those addressed in these projects.

To suggest the transaction mechanisms appropriate for a simple, clean and efficient implementation of an application, an understanding of the interactions in the application is imperative. ACTA is an attempt to provide a framework for capturing such interactions and reasoning about them.

CHAPTER 3

THE ACTA FORMAL FRAMEWORK

This chapter provides a concise yet complete introduction to ACTA and its formal underpinnings.

Section 3.1 provides some of the preliminary concepts underlying the ACTA formalism whereas Section 3.2 focuses on the concept of history which is central to the formalism. ACTA allows the specification of the effects of transactions on other transactions and also their effect on objects by means of constraints on histories. Inter-transaction dependencies, discussed in Section 3.3, forms the basis for the former while visibility of and conflicts between operations on objects, discussed in Section 3.4, form the basis for the latter. The final section, Section 3.5, shows how the ACTA formalism can be used to state correctness criteria related to extended transactions and objects.

We will use examples from various extended transaction models to illustrate the concepts. Complete specification of these models can be found in the following chapter, Chapter 4.

3.1 Preliminaries

3.1.1 Object Events

A database is the entity that contains all the shared objects in a system. A transaction accesses and manipulates the objects in the database by invoking operations specific to individual objects. The *state* of an object is represented by its contents. Each object has a type, which defines a set of operations that provide the only means to create, change and examine the state of an object of that type. It is assumed that an operation always produces an output (return value), that is,

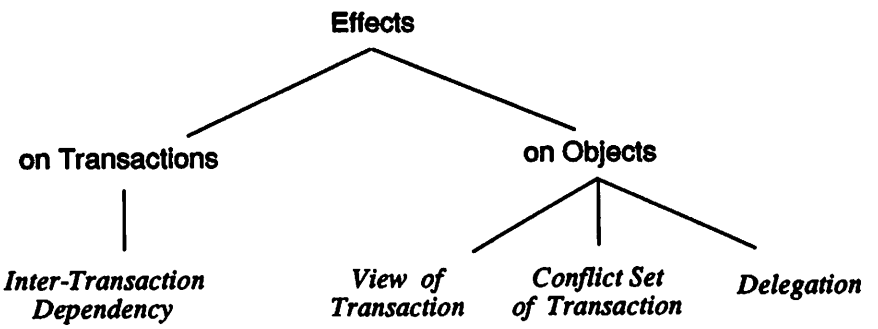

Figure 3.1 Dimensions of the ACTA Framework

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

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

DEFINITION 3.2: 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 3.3: 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.

[1]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."

Commit and *Abort* operations are defined on every object for every operation. Invoked operations that have neither committed nor aborted are termed *in progress* operations. Typically, an operation is committed only if the invoking transaction commits and it is aborted only if the invoking transaction aborts. However, it is conceivable that an extended transaction may commit only a subset of its operations on an object while aborting the rest. Furthermore, through delegation (see Section 3.4), a transaction other than the *event-invoker*, i.e., the transaction that invoked an operation, can be granted the responsibility to commit or abort the operation.

3.1.2 Significant Events

In addition to the invocation of operations on objects, transactions invoke transaction management primitives. For example, atomic transactions are associated with three transaction management primitives: Begin, Commit and Abort. The specific primitives and their semantics depend on the specifics of a transaction model. For instance, whereas the Commit by an atomic transaction implies that it is terminating successfully and that all of its effects on the objects should be made permanent in the database, the Commit of a subtransaction of a nested transaction implies that all of its effects on the objects should be made persistent and visible with respect to its parent and sibling subtransactions[2]. Other transaction management primitives include Spawn, found in the nested transaction model, and Split and Join, found in the split transaction model [82].

DEFINITION 3.4: 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.

[2]As shown in Section 3.4, in ACTA, the ability of a nested subtransaction to make its effect visible to its parent is specified by means of the notion of delegation.

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 3.5: *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 3.6: *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.

3.2 Histories and Conditions on Event Occurrences

Fundamental to ACTA is the notion of *history* [17] 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 invariant assertions about the histories generated by the transactions adhering to the particular model. Axioms can also be explicit *preconditions* or *postconditions* for operations and transaction management primitives. Consequently, the correctness properties of different transaction models can be expressed in terms of the properties of the histories produced by these models.

DEFINITION 3.7: 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 3.8: 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 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 3.9: 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 3.10: $(\epsilon \in H) \Rightarrow Condition_H$, 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 3.11: $Condition_H \Rightarrow (\epsilon \in H)$ 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 $H^{(ob)}$.

3.3 Effects of Transactions on Other Transactions

Dependencies provide a convenient way to specify and reason about the behavior of concurrent transactions and can be precisely expressed in terms of the significant events associated with the transactions.

DEFINITION 3.12: *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.

3.3.1 Types of Dependencies

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

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

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[3] (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

---

[3]Note that *serial dependency* as defined here constrains the execution of a transaction and has nothing to do with the notion of *serial dependency relations*, discussed in the previous chapter, which is a class of criteria for defining conflicts between operations.

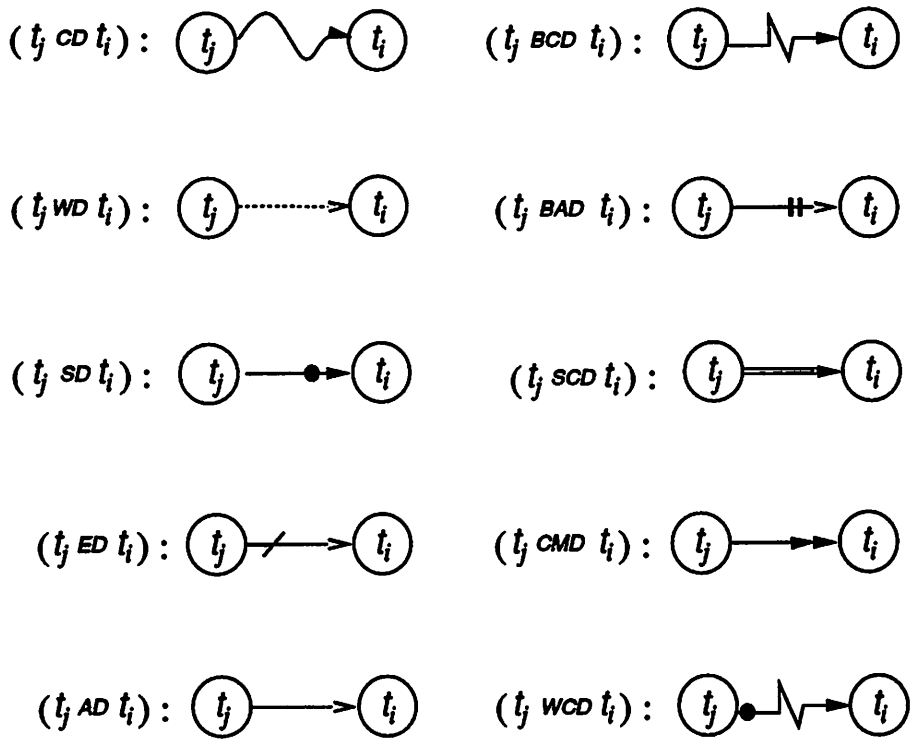

Figure 3.2 Inter-transaction Dependencies Graphs

of nested transactions [73]. Begin-on-commit dependency, begin-on-abort dependency and force-commit-on-abort dependency are useful for compensating transactions [58] and contingency transactions [21]. 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 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.

Besides the logical representation introduced above, inter-transaction dependencies can be expressed in a pictorial form as graphs whose vertices represent transactions and arcs of different shapes represent different dependencies. We refer

to such graphs as *dependency graphs*. Figure 3.2 shows the pictorial representation of some of the dependencies defined above. In general, dependency graphs can be more illustrative than the corresponding sets of axioms in expressing the structure of extended transactions, such as the explicit nesting structure of nested transactions. (As discussed in the next section, one source of dependencies is the structure of extended transactions.) Through dependency graphs, it is possible to capture both the static structure as well as the dynamics of the evolution of the structure of transactions. The structure of transactions evolves as significant events inducing inter-transaction dependencies occur. That is, a set of dependency graphs corresponds to $DepSet_{ct}$ which contains the inter-transaction dependencies developed until a point in time. In Chapter 5, based on such dependency graphs, we will examine the structural properties of a new transaction model which is derived by combining the nested transaction and split transaction models.

3.3.2 Source of Dependencies

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

3.3.2.1 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 ($(t_c \ WD \ t_p)$) and a parent establishing a commit dependency on its child ($(t_p \ CD \ t_c)$). Specifically, this is specified in terms of the postcondition of the Spawn event ($post(\mathsf{Spawn}_{t_p}[t_c])$):

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

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 Chapter 4 for a precise formal definition of nested transactions.) The commit dependency of the parent on its child is preserved if (1) the parent does not commit before its child terminates, or (2) the child aborts in case its parent commits first. 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. We prove this in Chapter 4.

Other hierarchically-structured transactions may define various relationships between a parent and its child transactions. For example, in the transaction model proposed in [21, 44] 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 Chapter 5). Child transactions may also have different dependencies with their parents if the transaction model supports various spawning or coupling modes [33]. Sibling transactions may also be interrelated in several ways. For example, components of a *saga* [45] can be paired according to a compensated-for/compensating relationship [58]. 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 4.7). In a similar fashion dependencies that occur in the presence of alternative transactions and contingency transactions [21] can also be specified.

3.3.2.2 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 [17] 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.

3.4 Objects and the Effects of Transactions on Objects

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

3.4.1 Conflicts between Operations and the Induced Dependencies

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

DEFINITION 3.13: 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 3.14: 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 dissertation, 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 [9] 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, 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:

---

[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*.

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

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 [90].

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 [41]. For instance, the condition $Notify_{t_j}[(t_i \; CD \; t_j)]$ 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 [85, 54, 100, 9, 28], and even to tailor them for cooperative environments.

3.4.2 Controlling Object Visibility

3.4.2.1 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 3.15: 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 follows.

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

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 3.16: 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.

3.4.2.2 Delegation

Traditionally, the invoker of an operation has the responsibility for committing or aborting the operation. In general, however, the operation invoker and the one committing the operation may be different.

DEFINITION 3.17: $ResponsibleTr(p_{t_i}[ob])$ identifies the transaction responsible for committing or aborting the operation $p_{t_i}[ob]$ with respect to the current history H_{ct}.

A transaction may *delegate* some of its responsibilities to another transaction. More precisely,

DEFINITION 3.18: $Delegate_{t_i}[t_j, p_{t_k}[ob]]$ denotes that t_i delegates to t_j the responsibility for committing or aborting operation $p_{t_k}[ob]$.

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

The transaction t_i that delegates an operation $p_{t_k}[ob]$ is called the *delegator* and the transaction t_j that is granted the responsibility for the operation is called the *delegatee*. Clearly, the delegator must be the transaction responsible for p_{t_k} at the time of the delegation. That is, the precondition for the event $Delegate_{t_i}[t_j, p_{t_k}[ob]]$ is that $ResponsibleTr(p_{t_k}[ob])$ is t_i. The postcondition will imply that $ResponsibleTr(p_{t_k}[ob])$ is t_j.

$$pre(Delegate_{t_i}[t_j, p_{t_k}[ob]]) \Rightarrow (ResponsibleTr(p_{t_k}[ob]) = t_i)$$

$$post(Delegate_{t_i}[t_j, p_{t_k}[ob]]) \Rightarrow (ResponsibleTr(p_{t_k}[ob]) = t_j)$$

This means that $ResponsibleTr(p_{t_i}[ob])$ 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.

$$(ResponsibleTr(p_{t_i}[op]) = t_j) \Rightarrow (p_{t_i} \in H_{ct}) \wedge$$
$$(((t_i = t_j) \wedge (\nexists t_k\ (Delegate_{t_i}[t_k, p_{t_i}[ob]] \in H_{ct}) \vee$$
$$(\exists t_m\ (Delegate_{t_m}[t_i, p_{t_i}[ob]] \in H_{ct}) \wedge$$
$$\nexists t_n\ (Delegate_{t_m}[t_i, p_{t_i}[ob]] \rightarrow Delegate_{t_i}[t_n, p_{t_i}[ob]]))))) \vee$$
$$((t_i \neq t_j) \wedge \exists t_m\ (Delegate_{t_m}[t_j, p_{t_i}[ob]] \in H_{ct}) \wedge$$
$$\nexists t_n\ (Delegate_{t_m}[t_j, p_{t_i}[ob]] \rightarrow Delegate_{t_j}[t_n, p_{t_i}[ob]])))$$

In words, the second clause says that either (1) t_j is t_i, in which case either $p_{t_i}[ob]$ was never delegated by t_i or it was delegated back to t_i by some transaction t_m after a number of delegations in between; or (2) t_j is different from t_i, in which case $p_{t_i}[ob]$ has been delegated to t_j and t_j has not subsequently delegated $p_{t_i}[ob]$ to any other transaction.

A precondition for any event that affects operations in the current history is that the transaction invoking the event is the transaction responsible for the affected operations. For example, a precondition for the event $Abort_{t_j}[p_{t_i}[ob]]$ is that $ResponsibleTr(p_{t_i}[ob])$ is t_j, while a precondition for the event $Commit_{t_j}[p_{t_i}[ob]]$ is that $ResponsibleTr(p_{t_i}[ob])$ is t_j. Hence, from now on, unless essential, we will drop the subscript t_j associated with the operation abort and commit events.

$$pre(Abort_{t_j}[p_{t_i}[ob]]) \Rightarrow (ResponsibleTr(p_{t_i}[ob]) = t_j)$$

$$pre(Commit_{t_j}[p_{t_i}[ob]]) \Rightarrow (ResponsibleTr(p_{t_i}[ob]) = t_j)$$

Delegation cannot occur after (1) the delegatee commits or aborts, and (2) the delegated operations have been committed or aborted.

$$(Delegate_{t_i}[t_j, p_{t_k}[ob]] \in H) \Rightarrow$$
$$((Delegate_{t_i}[t_j, p_{t_k}[ob]] \rightarrow \mathsf{Commit}_{t_j}) \vee$$
$$(Delegate_{t_i}[t_j, p_{t_k}[ob]] \rightarrow \mathsf{Abort}_{t_j}))$$

$$(Delegate_{t_i}[t_j, p_{t_k}[ob]] \in H) \Rightarrow$$
$$(((Delegate_{t_i}[t_j, p_{t_k}[ob]] \rightarrow Commit[p_{t_k}[ob]]) \vee$$
$$(Delegate_{t_i}[t_j, p_{t_k}[ob]] \rightarrow Abort[p_{t_k}[ob]])))$$

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 3.19: $AccessSet_t = \{p_{t_i}[ob] | ResponsibleTr(p_{t_i}[ob]) = t\}$; 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 Chapter 5.

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 *notifying* each other of their behavior. By being able to capture these aspects of transactions, the ACTA framework is applicable to cooperative environments.

3.5 Specifying Correctness Criteria in ACTA

In this section, we demonstrate the expressive power of the ACTA formalism introduced in the previous sections by capturing of various transactions' correctness criteria and properties of objects.

Before getting into specifics, let us first define the *transitive-closure* of a binary relation, let say $\mathcal{R}$, on transactions.

DEFINITION 3.20: Let $\mathcal{R}^*$ be the transitive-closure of $\mathcal{R}$; i.e.,

$(t_i \mathcal{R}^* t_k)$ if $[(t_i \mathcal{R} t_k) \vee \exists t_j \ (t_i \mathcal{R} t_j \wedge t_j \mathcal{R}^* t_k)]$.

3.5.1 Serializability

In traditional databases, serializability and in particular *conflict preserving serializability*, is the well-accepted criterion for concurrency control.

DEFINITION 3.21: Let C be a binary relation on transactions, and t_i and t_j be transactions.

$(t_i C t_j), t_i \neq t_j$ 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 3.22: A set of transactions T is *(conflict preserving) serializable* iff

$\forall t \in T \ \neg(t C^* t)$

3.5.2 Failure Atomicity

DEFINITION 3.23: 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.

3.5.3 Properties of Atomic Objects

Here, we define the correctness properties of objects.

DEFINITION 3.24: An object *ob* behaves *correctly* iff

$\forall t_i, t_j, t_i \neq t_j \; \forall p, q$

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

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 interactions between operations. Similarly, such dependencies can be defined for deferred interactions.

DEFINITION 3.25: An object *ob* behaves *serializably* iff

$$\forall t \forall p \; (\textit{Commit}[p_t[ob]] \in H^{(ob)}) \Rightarrow \neg(tC^*t).$$

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

DEFINITION 3.26: An object *ob* is *atomic* if *ob* behaves *correctly* and *serializably*.

3.5.4 Predicatewise Serializability

Predicatewise serializability has been proposed in [57, 60] as the correctness criterion for concurrency control in databases in which consistency constraints are in a conjunctive normal form. In such cases, consistency constraints can be maintained by requiring serializability only with respect to objects which relate to a disjunctive clause.

DEFINITION 3.27: Suppose the consistency constraints that appear in a disjunctive clause relate to objects $D_i \subseteq DB$, where DB is the database.

DEFINITION 3.28: Let C_{D_i} be a binary relation on transactions, and t_i and t_j be transactions.

$(t_i C_{D_i} t_j), t_i \neq t_j$ if

$\exists ob \in D_i \; \exists p, q \; (conflict(p_{t_i}[ob], q_{t_j}[ob]) \wedge (p_{t_i}[ob] \rightarrow q_{t_j}[ob]))$

DEFINITION 3.29: A set of transactions T is *predicatewise serializable* iff

$\forall t \in T \; \forall D_i \; \neg(t C^*_{D_i} t)$

3.5.5 Cooperative Serializability

Just as predicatewise serializability is defined with respect to objects which share a predicate, we can define *cooperative serializability* with respect to a set of transactions which maintain some consistency properties [68]. Transactions in a set could be the components of an extended transaction, or transactions collaborating over some objects while maintaining the consistency of the objects. In such cases, consistency can be maintained if other transactions which do not belong to the set, are serialized with respect to all the transactions in the set. In other words, the set of cooperative transactions becomes the unit of serializability.

DEFINITION 3.30: T_c be a set of cooperative transactions. Let C_c be a binary relation on transactions, and t_i, t_j and t_k be transactions.

$(t_i C_c t_j), t_i \neq t_j, t_i \neq t_k, t_j \neq t_k$ if

$$\begin{aligned}
&\exists ob \; \exists p, q \; ((t_i \notin T_c, t_j \notin T_c (conflict(p_{t_i}[ob], q_{t_j}[ob]) \wedge (p_{t_i}[ob] \rightarrow q_{t_j}[ob]))) \vee \\
&(t_i \notin T_c, t_j \in T_c, t_k \in T_c (conflict(p_{t_i}[ob], q_{t_k}[ob]) \wedge (p_{t_i}[ob] \rightarrow q_{t_k}[ob]))) \vee \\
&(t_i \in T_c, t_j \notin T_c, t_k \in T_c (conflict(p_{t_k}[ob], q_{t_j}[ob]) \wedge (p_{t_k}[ob] \rightarrow q_{t_j}[ob])))))
\end{aligned}$$

In this definition, the first clause expresses how a dependency between two transactions which do not belong to the same set is directly established when they invoke conflicting operations on a shared object. This is similar to the clause in the classical definition of (conflict preserving) serializability. The other two clauses reflect the fact that when a transaction establishes a dependency with another transaction, the

same dependency is established between all the transactions in their corresponding cooperative transactions sets. These clauses can be viewed as expressions of the development of dependencies between transaction sets.

DEFINITION 3.31: A set of transactions T is *cooperative serializable* iff

$$\forall t \in T \ \neg(tC_c^*t)$$

3.5.6 Quasi Serializability

Quasi Serializability has been proposed in [36, 37] as a correctness criterion for maintaining transaction consistency in multidatabases, i.e., heterogeneous distributed databases. In these systems, transactions can either execute on a single site (called *local* transactions), or can execute on multiple sites (called *global* transactions).

A set of of local and global transactions is *quasi serializable* if (1) all local histories are (conflict preserving) serializable, and (2) there exists a total order of all global transactions g_i and g_j where g_i precedes g_j in the order and all g_i's operations precede g_j's operations in all local histories in which they both appear.

DEFINITION 3.32: Let L_i be the set of transactions executing on node i.

DEFINITION 3.33: Let $\mathcal{R}$ be a binary relation on a set of global transactions G, and g_i and g_j in G.

$(g_i \mathcal{R} g_j), g_i \neq g_j$ if

$$\exists L_k \ \exists t_i \in L_k, t_i \text{ component of } g_i \ \exists t_j \in L_k, t_j \text{ component of } g_j \ \exists ob \ \exists p, q$$
$$(\mathit{conflict}(p_{t_i}[ob], q_{t_j}[ob]) \wedge (p_{t_i}[ob] \rightarrow q_{t_j}[ob]))$$

DEFINITION 3.34: A set of local and global transactions T is *quasi serializable* if

1. $\forall i \ \forall t \in L_i \ (\mathsf{Commit}_t \in H) \Rightarrow \neg(tC^*t)$,
2. $\forall g \in G \ (\mathsf{Commit}_g \in H) \Rightarrow \neg(g\mathcal{R}^*g)$.

3.5.7 Setwise Failure Atomicity

Setwise failure atomicity is a generalization of failure atomicity which is suitable for set of transactions or extended transactions composed of a set of transactions; i.e., an element of the set commits iff every element commits.

DEFINITION 3.35: A set of transactions T is *setwise failure atomic* if

1. $\exists t_i \in T\ \exists ob\ \exists p\ (Commit_{t_i}[p_{t_i}[ob]] \in H) \Rightarrow$
   $\forall t_j \in T\ \forall ob'\ \forall q\ ((q_{t_j}[ob'] \in H) \Rightarrow (Commit_{t_j}[q_{t_j}[ob']] \in H))$,
2. $\exists t_i \in T\ \exists ob\ \exists p\ (Abort_{t_i}[p_{t_i}[ob]] \in H) \Rightarrow$
   $\forall t_j \in T\ \forall ob'\ \forall q\ ((q_{t_j}[ob'] \in H) \Rightarrow (Abort_{t_j}[q_{t_j}[ob']] \in H))$,

In words, condition 1 states that if an operation invoked by a transaction t belonging to T is committed, all the operations invoked by transactions in T are committed. Condition 2 states that if an operation invoked by a transaction t belonging to T is aborted, all the operations invoked by transactions in T are aborted.

3.5.8 Quasi Failure Atomicity

Failure atomicity requires that all operations are committed or aborted by the invoking transaction. Thus, failure atomicity does not apply in the presence of delegation. However, in the presence of delegation, the following situation is possible: All the operations performed by a transaction t and not delegated to another transaction, say t_i, are committed (aborted) by t, and all the delegated operations are committed (aborted) by t_i. In this case, all or none of t's operations have been executed and hence, we called it *quasi failure atomicity.*

DEFINITION 3.36: Transaction t is *quasi failure atomic* if

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

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 by t or some other transaction, say t_i, all the operations invoked by t are committed by t or some other transaction, say t_j. The "none" clause is captured by condition 2 which states that if an operation invoked by a transaction t is aborted by t or some other transaction t_i, all the operations invoked by t are aborted by t or some t_j.

3.6 Conclusion

ACTA is based on First Order Predicate Calculus with a precedence relation and also on a small set of powerful building blocks. In this chapter, we formally introduced the building blocks of the ACTA framework, namely, *inter-transaction dependencies*, *conflict relations* between operations, *view* of a transaction, *conflict set* of a transaction and the notion of *delegation*. Central to ACTA is the notion of history of object event and significant event invocations by transactions. In ACTA, the definition of a transaction model consists of a set of axioms which are either invariant assertions about the histories produced by the transactions of the particular model or explicit preconditions or postconditions of operation or transaction management primitives. Also, the correctness properties of different transaction models can be expressed in terms of the histories produced by these models. There are three ways that an occurrence of an event in a history can be affected: (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. Based on these, dependencies between concurrent transactions, and conflict relations between operation on objects can been defined in ACTA.

Subsequently, we discussed how ACTA can capture the (extended) functionality of a transaction model (1) by allowing the specification of significant events beyond commit and abort, (2) by allowing the specification of arbitrary transaction structures in terms of dependencies involving any significant event, (3) by supporting finer grain visibility for objects in the database by means of views, conflict sets and the notion of delegation, (4) and by facilitating object-specific and transaction-specific semantic-based concurrency control.

Finally, we illustrated the use of the ACTA formalism by stating different correctness criteria of (extended) transactions and properties of objects in ACTA. In the following chapters, these will be used in the formal definition of various transaction models and will form the basis for reasoning about the properties of these models.

CHAPTER 4

SPECIFICATION AND ANALYSIS OF EXTENDED TRANSACTION MODELS

In this chapter, we further examine the expressive power of ACTA, the formal framework introduced in the previous chapter. This is achieved by formally specifying six representative (extended) transaction models and showing how these specifications can be used to reason about the correctness properties of some of these models. The properties cover a broad spectrum of possible combinations of the transaction properties of visibility, consistency, recovery and permanence — from the restrictive ACIDity properties of atomic transactions to the complex sagas. ACTA has also been successfully used for characterizing the structure and behavior of extended transactions other than the ones discussed in this chapter [21, 68].

The chapter begins by defining the *fundamental axioms* of transactions which are applicable to all transaction models. Each of the following sections is devoted to a particular transaction model: Section 4.2 to atomic transactions, Section 4.3 to distributed transactions, Section 4.4 to nested transactions, Section 4.5 to split transactions and joint transactions, Section 4.6 to recoverable communicating actions, and Section 4.7 to sagas. The atomic transaction model is included because atomic transactions constitute the base model and form the basis for many of the extended transaction models.

4.1 Fundamental Axioms of Transactions

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 significant events, called

initiation events, that can be invoked to initiate the execution of the transaction, and a set of significant events, called *termination events* that can be invoked to terminate the execution of the transaction. 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.

Notation: Recall from Section 3.1.2 that SE_t denotes the set of significant events that pertain to a transaction t, IE_t denotes the set of initiation events associated with t and TE_t denotes the set of termination events associated with t. As we shall see, SE_t, TE_t and IE_t depend on the transaction model of t. 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 4.1: *Fundamental Axioms of Transactions*
Let t be a transaction and H^t the projection of the history H with respect to t.

I. $\forall \alpha, \beta \in IE_t \; (\alpha \in H^t) \Rightarrow (\beta \notin H^t)$

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

III. $\forall \gamma, \delta \in TE_t \; (\gamma \in H^t) \Rightarrow (\delta \notin H^t)$

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

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

4.2 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. (Refer to Section 3.5 for the definition of these two properties in ACTA.)

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

DEFINITION 4.2: *Axiomatic definition of Atomic Transactions*

t denotes an atomic transaction.

1. $SE_t = \{\text{Begin, Commit, Abort}\}$
2. $IE_t = \{\text{Begin}\}$
3. $TE_t = \{\text{Commit, Abort}\}$
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. $(\text{Commit}_t \in H) \Rightarrow \neg(tC^*t)$.
9. $\exists ob\ \exists p\ (Commit_t[p_t[ob]] \in H) \Rightarrow (\text{Commit}_t \in H)$
10. $(\text{Commit}_t \in H) \Rightarrow \forall ob\ \forall p\ ((p_t[ob] \in H) \Rightarrow (Commit_t[p_t[ob]] \in H))$
11. $\exists ob\ \exists p\ (Abort_t[p_t[ob]] \in H) \Rightarrow (\text{Abort}_t \in H)$
12. $(\text{Abort}_t \in H) \Rightarrow \forall ob\ \forall p\ ((p_t[ob] \in H) \Rightarrow (Abort_t[p_t[ob]] \in H))$

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
   $((\text{Begin}_t \in H) \Rightarrow \neg(\text{Begin}_t \rightarrow \text{Begin}_t))$ [Axiom I],
2. only an initiated transaction can commit or abort
   $((\text{Commit}_t \in H) \Rightarrow (\text{Begin}_t \rightarrow \text{Commit}_t)$, and
   $(\text{Abort}_t \in H) \Rightarrow (\text{Begin}_t \rightarrow \text{Abort}_t))$ [Axiom II], and
3. an atomic transaction cannot be committed after it has been aborted
   $((\text{Commit}_t \in H) \Rightarrow ((\text{Abort}_t \notin H) \wedge \neg(\text{Commit}_t \rightarrow \text{Commit}_t)))$, and vice versa
   $((\text{Abort}_t \in H) \Rightarrow ((\text{Commit}_t \notin H) \wedge \neg(\text{Abort}_t \rightarrow \text{Abort}_t)))$ [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 (defined in Section 3.5.3). 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 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 [99]. 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.

LEMMA 4.1: If t is an atomic transaction, then t is *failure atomic.*

PROOF: For t to be failure atomic, t must satisfy the two conditions of the definition of failure atomicity [Definition 3.23]. Given that there is no delegation,

1. Condition 1 (the "all" clause) is derived from Axioms 9 and 10.
2. Condition 2 (the "none" clause) is derived from Axioms 11 and 12. □

THEOREM 4.1: Atomic transactions have the following properties:

1. If t is an atomic transaction, t is *failure atomic,*
2. A set of committed atomic transactions T is *serializable.*

PROOF: The first property corresponds to lemma 4.1. The second property directly follows from the fact that (1) the C relation is established between atomic transactions with conflicting operations [Axiom 7] and (2) Axiom 8 satisfies to the criterion of serializable executions [definition 3.22]. □

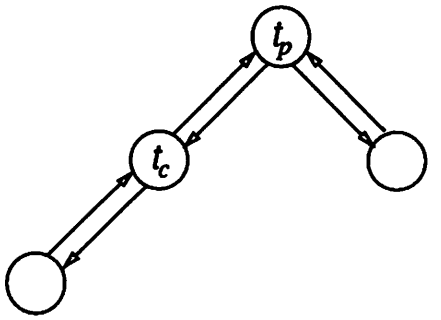

Figure 4.1 Structure of Distributed Transactions

DEFINITION 4.3: An atomic transaction management scheme is *correct* if it conforms to Definition 4.2.

4.3 Distributed Transactions

In most existing distributed databases systems, transactions are decomposed into subtransactions in order to invoke operations on objects which are physically distributed in a computer network. Thus, typically, each subtransaction executes on a different site in the network. A subtransaction can further be decomposed into other subtransactions and thus, distributed transactions establish a hierarchical structure. Each subtransaction is atomic. Subtransactions can commit only if the whole distributed transaction to which they belong, commits. The abortion of a subtransaction causes the abortion of the whole distributed transaction.

Components of distributed transactions are distinguished into two types, namely, *root transactions* and *(distributed) subtransactions.* Here are the semantics of both of these transaction types as expressed in ACTA.

DEFINITION 4.4: *Axiomatic definition of Distributed Transactions*

t_0 denotes the root transaction.

t_c denotes a subtransaction of t_p.

t denotes a root t_0 or a subtransaction t_c.

1. SE_{t_0} = {Begin, Spawn, Commit, Abort}
2. IE_{t_0} = {Begin}
3. TE_{t_0} = {Commit, Abort}

4. SE_{t_c} = {Spawn, Commit, Abort}
5. IE_{t_c} = {Spawn}
6. TE_{t_c} = {Commit, Abort}
7. t satisfies the fundamental Axioms I to IV
8. $\forall ob\ \exists p\ (p_t[ob] \in H) \Rightarrow (ob\ is\ atomic)$
9. $View_t = H_{ct}$
10. $ConflictSet_t = \{p_{t'}[ob] \mid t' \neq t, Inprogress(p_{t'}[ob])\}$
11. $(\text{Commit}_t \in H) \Rightarrow \neg(tC^*t)$
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. $post(\text{Spawn}_{t_p}[t_c]) \Rightarrow (((t_c\ \mathcal{AD}\ t_p) \in DepSet_{ct}) \wedge ((t_p\ \mathcal{AD}\ t_c) \in DepSet_{ct}))$

Axioms 1 and 4 state that root transactions and subtransactions are associated with different significant events. In spite of the fact that the two transactions types are initiated by different events [Axioms 2 and 5], i.e., root transactions by Begin and subtransactions by Spawn, the semantics of both transaction types are similar to atomic transactions [Axioms 3, 6–15].

Axiom 16 captures the parent/child relationship between the spawning and spawned transactions through symmetric abort dependencies. The symmetric abort dependencies ensure that either all subtransactions commit or the whole distributed computation aborts. The relationship is established at the time a subtransaction is spawned, and hence, it is specified in terms of the postconditions of the Spawn event.

DEFINITION 4.5: Let $T_d = \{t_0, t_1, ..., t_n\}$ be a distributed transaction where:

1. t_0 is the root transaction, and
2. $\forall t \in T_d, t \neq t_0\ \exists t_i \in T_d\ (\text{Spawn}_{t_i}[t] \in H)$.

LEMMA 4.2: Let T_d be a distributed transaction. If $t \in T_d$, t is *failure atomic.*

PROOF: Given Axioms 12–15, proof is similar to that of lemma 4.1 (the failure atomicity property of atomic transactions) □

LEMMA 4.3: If T_d is a distributed transaction, T_d is *setwise failure atomic.*

PROOF: We will show that T_d satisfies the definition of setwise failure atomicity [definition 3.35].

1. Given that each t_i in T_d can either commit or abort [Axioms 3 and 6], Axiom 16 implies that either all transactions commit or all abort:
   i. $\exists t_i \in Td\ (\mathsf{Commit}_{t_i} \in H) \Rightarrow \forall t_j \in T_d\ (\mathsf{Commit}_{t_j} \in H)$
   ii. $\exists t_i \in Td\ (\mathsf{Abort}_{t_i} \in H) \Rightarrow \forall t_j \in T_d\ (\mathsf{Abort}_{t_j} \in H)$
2. Given 1(*i*) and Axioms 12 and 13, if an operation invoked by a t_i in T_d is committed, then all operations invoked by transactions in T_d are committed:
   $\exists t_i \in T_d\ \exists ob\ \exists p\ (Commit_{t_i}[p_{t_i}[ob]] \in H) \Rightarrow$
   $\forall t_j \in T_d\ \forall ob'\ \forall q\ ((q_{t_j}[ob'] \in H) \Rightarrow (Commit_{t_j}[q_{t_j}[ob']] \in H))$.
3. Given 1(*ii*) and Axioms 14 and 15, if an operation invoked by a t_i in T_d is aborted, then all operations invoked by transactions in T_d are aborted:
   $\exists t_i \in T_d\ \exists ob\ \exists p\ (Abort_{t_i}[p_{t_i}[ob]] \in H) \Rightarrow$
   $\forall t_j \in T_d\ \forall ob'\ \forall q\ ((q_{t_j}[ob'] \in H) \Rightarrow (Abort_{t_j}[q_{t_j}[ob']] \in H))$.
4. Thus, given that there is no delegation, from (2) and (3), T_d satisfies Definition 3.35. □

THEOREM 4.2: Distributed transactions have the following properties:

1. If T_d is a distributed transaction, T_d is *setwise failure atomic,*
2. A set of committed distributed transactions S is *serializable.*

PROOF: Property 1 follows from lemma 4.3. Given Axioms 8 and 11, the serializability property of distributed transactions (property 2) directly follows from the definition of serializable transactions [Definition 3.22]. □

DEFINITION 4.6: A distributed transaction management scheme is *correct* if it conforms to Definition 4.4.

4.4 Nested Transactions

In the Nested Transaction model, e.g. [73], transactions are composed of subtransactions or child transactions designed to localize failures within a transaction

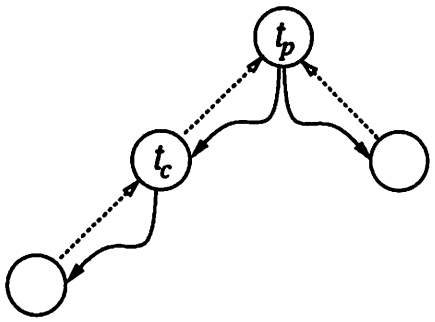

Figure 4.2 Structure of Nested Transactions

and to exploit parallelism within transactions. A subtransaction can be further decomposed into other subtransactions, and thus, a transaction may expand in a hierarchical manner. Subtransactions execute atomically with respect to their siblings and other non-related transactions and are failure atomic with respect to their parent. They can abort independently without causing the abortion of the whole transaction.

A subtransaction can potentially access any object that is currently accessed by one of its ancestor transactions. In addition, any object in the database is also potentially accessible to the subtransaction. When a subtransaction commits, the objects modified by it are made accessible to its parent transaction. However, the effects on the objects are made permanent in a database only when the root transaction commits.

Now, let us define nested transactions using the ACTA formalism. Here, we need a definition of serializability which takes into account the presence of delegation.

DEFINITION 4.7: Let C_N be a binary relation on transactions, and t_i and t_j be transactions.

$(t_i C_N t_j), t_i \neq t_j$ if

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

This definition extends the definition of the C relation [Definition 3.21] to include the dependencies due to the delegated objects. (To see that C_N is a generalization of C,

consider the case in which delegation does not occur. In the absence of delegation, $t_m = t_i$ and $t_n = t_j$.) In this way, by substituting C_N for C in the definition of serializability [Definition 3.22], transactions are serialized with respect to both their invoked operations and the operations for which they are responsible, i.e., the operation delegated to them.

DEFINITION 4.8: A set of transactions T is *(conflict preserving) serializable* iff

$$\forall t \in T \; \neg(tC_N^*t)$$

Also, there is a need to revisit the definition of failure atomicity in face of delegation. As in the case of serializability, this leads to a definition that is a generalization of Definition 3.23. (To see this, recall that $(ResponsibleTr(q_{t_j}[ob']) = t) \Rightarrow (q_{t_j}[ob'] \in H)$ [Section 3.4]).

DEFINITION 4.9: Transaction t is *failure atomic* if

1. $\exists ob \; \exists p \; \exists t_i \; (Commit_t[p_{t_i}[ob]] \in H) \Rightarrow$
   $\forall ob' \; \forall q \; \forall t_j \; ((ResponsibleTr(q_{t_j}[ob']) = t) \Rightarrow (Commit_t[q_{t_j}[ob']] \in H))$,
2. $\exists ob \; \exists p \; \exists \; (Abort_t[p_{t_i}[ob]] \in H) \Rightarrow$
   $\forall ob' \; \forall q \; \forall t_j \; ((ResponsibleTr(q_{t_j}[ob']) = t) \Rightarrow (Abort_t[q_{t_j}[ob']] \in H))$,

According to this general definition of failure atomicity, a transaction t is failure atomic if either "all" or "none" of the operations for which the transaction t is responsible are committed.

The nested transaction model supports two types of transactions, namely, *root transactions* and *nested subtransactions*, which are associated with different significant events [Axioms 1 and 4]. The semantics of root transactions are similar to atomic transactions [Axioms 7–15]. The Abort event has the same semantics for both transaction types which are similar to those of the Abort in atomic transactions [Axioms 13 and 14]. However, the semantics of the Commit event are different for each transaction type. In the case of a root transaction, Commit has the semantics of the Commit event in atomic transactions [Axioms 10–12]. In contrast, when a subtransaction commits, through *delegation*, the operations in its access set are made persistent and visible only to its parent transaction [Axiom 18].

Spawn is used to initiate a new subtransaction. The Spawn event establishes a parent/child relationship between the spawning and spawned transactions. This relationship is reflected by the weak-abort dependency $\mathcal{WD}$ and commit dependency $\mathcal{CD}$ between the related transactions [Axiom 17]. The ability of a subtransaction to invoke operations without conflicting with the operations of its ancestor transactions is expressed by excluding all the operations performed by its ancestors from the conflict set of the subtransaction [Axiom 16].

DEFINITION 4.10: *Axiomatic definition of Nested Transactions*

t_0 denotes the root transaction.

t_p denotes a root or a subtransaction.

t_c denotes a subtransaction of t_p.

1. $SE_{t_0} = \{\text{Begin, Spawn, Commit, Abort}\}$
2. $IE_{t_0} = \{\text{Begin}\}$
3. $TE_{t_0} = \{\text{Commit, Abort}\}$
4. $SE_{t_c} = \{\text{Spawn, Commit, Abort}\}$
5. $IE_{t_c} = \{\text{Spawn}\}$
6. $TE_{t_c} = \{\text{Commit, Abort}\}$
7. t_p satisfies the fundamental Axioms I to IV
8. $\forall ob\ \exists p\ (p_{t_p}[ob] \in H) \Rightarrow (ob\ is\ atomic)$
9. $View_{t_p} = H_{ct}$
10. $(\text{Commit}_{t_p} \in H) \Rightarrow \neg(t_p C_N^* t_p)$
11. $\exists ob\ \exists p\ \exists t\ (Commit_{t_0}[p_t[ob]] \in H) \Rightarrow (\text{Commit}_{t_0} \in H)$
12. $(\text{Commit}_{t_0} \in H) \Rightarrow$
    $\forall ob\ \forall p\ \forall t\ ((ResponsibleTr(p_t[ob]) = t_0) \Rightarrow (Commit_{t_0}[p_t[ob]] \in H))$
13. $\exists ob\ \exists p\ \exists t\ (Abort_{t_p}[p_t[ob]] \in H) \Rightarrow (\text{Abort}_{t_p} \in H)$
14. $(\text{Abort}_{t_p} \in H) \Rightarrow$
    $\forall ob\ \forall p\ \forall t\ ((ResponsibleTr(p_t[ob]) = t_p) \Rightarrow (Abort_{t_p}[p_t[ob]] \in H))$
15. $ConflictSet_{t_0} = \{p_t[ob] \mid t \neq t_0, Inprogress(p_t[ob])\}$
16. $ConflictSet_{t_p} = \{p_t[ob] \mid t \neq t_p, t \notin Ancestors(t_p), Inprogress(p_t[ob])\}$
17. $post(\text{Spawn}_{t_p}[t_c]) \Rightarrow (((t_c\ \mathcal{WD}\ t_p) \in DepSet_{ct}) \wedge ((t_p\ \mathcal{CD}\ t_c) \in DepSet_{ct}))$
18. $(\text{Commit}_{t_c} \in H) \Leftrightarrow (Delegate_{t_c}[t_p, AccessSet_{t_c}] \in H)$

19. $\forall t \in Descendants(t_p)\ \forall ob\ \forall q\ pre(q_{t_p}[ob]) \Rightarrow$
$(\nexists p\ ((p_t[ob] \in H_{ct}) \wedge conflict(p_t[ob], q_{t_p}[ob]) \wedge (Responsible(p_t[ob]) \neq t_p))$

$Ancestors(t)$ is the set of all ancestors of t whereas $Descendants(t)$ is the set of all descendants of t.

Axiom 18 which together with Axiom 10 define the semantics of the Commit event of subtransactions, clearly specifies that the commitment of a subtransaction does not imply the commitment of its operations and the operations that it is responsible for.

Axiom 19 states that given transaction t and its ancestor t_p and conflicting operations p and q, t_p cannot invoke q after t invokes p and before p is delegated to t_p. In the absence of this restriction, it would be possible for an ancestor t_p of a transaction t to develop an abort dependency on t (t_p $\mathcal{AD}$ t) by invoking an operation that is return-value dependent on a preceding operation invoked by t. In such a case in which a parent transaction develops an abort dependency on its child, if the child aborts, the parent also aborts. This means that it would be possible for a subtransaction to cause the abortion of its parent and possibly of the whole nested transaction (if the parent happens to be the root transaction). But this violates the property of nested transactions that localizes failures by allowing a subtransaction to abort independently without causing the abortion of the whole transaction.

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

LEMMA 4.4: A nested subtransaction t_c is *quasi failure atomic.*

PROOF: It can be shown by induction on the depth of the hierarchy where a root transaction is at depth 0.

Basic step: Let t_c be a subtransaction at depth 1. That is, t_c is a child of the root.

1. If an operation invoked by t_c aborts, due to Axioms 13 and 14, all operations invoked by t_c are aborted.
2. If t_c commits, due to Axiom 18, all operations invoked by t_c are delegated to the root. Since the root is failure atomic [Axioms 11–14], either all delegated operations will be aborted by the root or all will be committed.

Thus, t_c satisfies the definition of quasi failure atomicity [definition 3.36].

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

1. If an operation invoked by t_c aborts, due to axioms 13 and 14, all the operations invoked by t_c will be aborted.
2. If t_c commits, due to Axiom 18, all operations invoked by t_c are delegated to t_p. Since t_p is quasi failure atomic [induction hypothesis] either all delegated operations will be aborted by t_c or one of its ancestors or all will be committed.

Thus, t_c is quasi failure atomic. □

Although Axioms 7, 8, and 10 would be sufficient to ensure the serializability of atomic transactions, they are not in the case of nested transactions because of Axiom 16 which allows dependencies between a parent transaction and its children to be ignored. Thus, a parent and a child transaction are not serializable.

LEMMA 4.5: A set of committed transactions T containing a nested subtransaction t_c, all its siblings and other non-related transactions is *serializable*.

This informally states that a t_c is serializable relative to its siblings and other non-related transactions.

PROOF: Given Axioms 8 and 16, a C_N relation is established between t_c, its siblings and the other non-related transactions in T when they invoked conflicting operations. Also, a C_N relation is established between them if they are responsible for conflicting operations. Thus, given Axiom 10, T satisfies the serializability criterion [Definition 4.8]. □

LEMMA 4.6: *No Orphan Commits*

Let H be a history of a nested transaction, t_p and t_c be transactions where t_p be the parent of t_c.

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

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

PROOF: This lemma is derived by logically rewriting $(t_p \ CD \ t_c)$ and $(t_c \ WD \ t_p)$ induced by Axiom 17.

1. $(t_p \; \mathcal{CD} \; t_c) \Leftrightarrow ((\text{Commit}_{t_p} \in H) \Rightarrow ((\text{Commit}_{t_c} \in H) \Rightarrow (\text{Commit}_{t_c} \rightarrow \text{Commit}_{t_p}))) \Leftrightarrow$
   $(\neg(\text{Commit}_{t_p} \in H) \vee \neg(\text{Commit}_{t_c} \in H) \vee (\text{Commit}_{t_c} \rightarrow \text{Commit}_{t_p})) \Leftrightarrow$
   $(\neg(\neg(\text{Commit}_{t_p} \in H) \vee (\text{Commit}_{t_c} \rightarrow \text{Commit}_{t_p})) \Rightarrow \neg(\text{Commit}_{t_c} \in H))$
   [given that H is a complete history and Axioms 6 and 7] $\Leftrightarrow$
   $(((\text{Commit}_{t_p} \in H) \wedge \neg(\text{Commit}_{t_c} \rightarrow \text{Commit}_{t_p})) \Rightarrow (\text{Abort}_{t_c} \in H))$
2. $(t_c \; \mathcal{WD} \; t_p) \Leftrightarrow ((\text{Abort}_{t_p} \in H) \Rightarrow (\neg(\text{Commit}_{t_c} \rightarrow \text{Abort}_{t_p}) \Rightarrow (\text{Abort}_{t_c} \in H))) \Leftrightarrow$
   $(((\text{Abort}_{t_p} \in H) \wedge \neg(\text{Commit}_{t_c} \rightarrow \text{Abort}_{t_p})) \Rightarrow (\text{Abort}_{t_c} \in H))$
3. From (1) and (2),
   $(((\text{Commit}_{t_p} \in H) \wedge \neg(\text{Commit}_{t_c} \rightarrow \text{Commit}_{t_p})) \vee$
   $((\text{Abort}_{t_p} \in H) \wedge \neg(\text{Commit}_{t_c} \rightarrow \text{Abort}_{t_p}))) \Rightarrow (\text{Abort}_{t_c} \in H).$ □

THEOREM 4.3: Nested transactions have the following properties:

1. A set of committed nested transactions T is *serializable.*
2. Operations are committed only by root transactions:
   $\forall ob \; \forall p \; \forall t \; ((p_t[ob] \in H) \wedge (Commit_{t_r}[p_t[ob]] \in H)) \Rightarrow (t_r$ is a root transaction)
3. If a root transaction t_0 aborts, all operations performed by t_0 and its descendants abort:
   $(\text{Abort}_{t_0} \in H) \Rightarrow$
   $\forall t, (t = t_0 \vee t \in Descendants(t_0)) \; \forall ob \; \forall p \; ((q_t[ob] \in H) \Rightarrow (Abort[p_t[ob]] \in H))$

PROOF: Property 1 requires that the effects of operations of root transactions in T are committed in the database in a serializable fashion. Property 2 requires that only root transactions can commit operations in the database. These properties are derivable from the semantics of delegation (note that once delegation is performed by a subtransaction when it commits, the responsibility for committing its operations is given to its parent), the semantics of atomic objects, the *no orphan commits* lemma, and Axiom 10.

Property (3) which says that if a nested transaction aborts, the operations invoked by its root and its subtransactions are all aborted, follows from the *no orphan commits* lemma, the failure atomicity property of the root transaction, and the quasi failure atomicity property of subtransactions. □

DEFINITION 4.11: A nested transaction management scheme is *correct* if it conforms to definition 4.10.

4.4.1 Comparing Nested and Distributed Transactions

The differences between distributed transactions and nested transactions are not immediately clear based on their informal descriptions. In fact, based on their informal description, one may believe that the two transaction models are identical because of the names of their events, e.g., Spawn, Commit and Abort, and the hierarchical structure of the transactions. However, given their axiomatic definitions [Definitions 4.4 and 4.10], it becomes clear that the corresponding events associated with the subtransactions have different semantics in each model, yielding different transaction properties. For instance, distributed subtransactions and nested subtransactions have different visibility properties since they are associated with different conflict sets. The conflict set of a nested subtransaction, by not containing the operations performed by its ancestors [Axiom 16], has a smaller size than the conflict set of a distributed subtransaction which includes all in-progress operations invoked by other transactions [Axiom 10]. Hence, nested subtransactions potentially are involved in fewer conflicts than their distributed counterparts which in turn means that nested transactions allow higher degree of visibility than distributed transactions.

Distributed transactions and nested transactions also have different permanence properties. Distributed subtransactions can commit their operations making their effects permanent in the database [Axioms 12 and 13] whereas nested subtransactions cannot. Only the root of a nested transaction can commit an operation [Theorem 4.3].

The hierarchical structure of distributed transactions is different from that of nested transactions as this is reflected in their respective dependency graphs which involve different edges (see Figures 4.1 and 4.2). (The dependency graph of distributed transactions is derived from Axiom 16 whereas that of nested transactions is derived from Axiom 17.) These structural differences between distributed transactions and nested transactions result in differences in their recovery semantics.

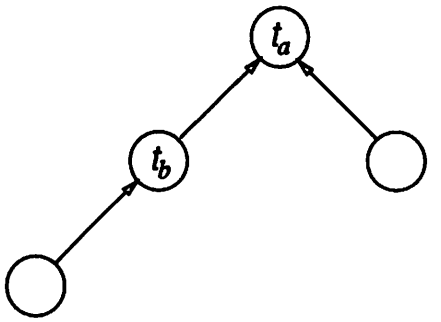

Figure 4.3 Structure of Split Transactions

First, the pair of commit and weak-abort dependencies capturing the parent/child relationship in nested transactions allows sufficient flexibility for nested transactions to tolerate partial failures. In contrast, the symmetric abort dependencies capturing the parent/child relationship in distributed transactions does not allow such flexibility. Thus, distributed transactions cannot tolerate any failure. Secondly, distributed subtransactions are failure atomic [Lemma 4.2] but nested subtransactions are not. They are quasi failure atomic [Lemma 4.4].

4.5 Split and Joint Transactions

In the Split Transaction model [82], it is possible for a transaction t_a to split into two transactions, t_a and t_b and for two transactions t_a and t_b to join into one joint transaction t_b. For simplicity, we treat here the split transactions and joint transactions as two distinct transaction models. The former is discussed in the next subsection and the latter in Section 4.5.2.

4.5.1 Split Transactions

In the split transaction model, a transaction t_a can split into transactions t_a and t_b. At the time of the split, operations invoked by t_a up to the split can be divided between t_a and t_b making each responsible for committing and aborting those operations assigned to them. In order to facilitate further data sharing between t_a and t_b, operations which remain the responsibility of t_a may be designated as not conflicting with operations invoked by t_b after the split, and hence, t_b can view the

effects of these operations. Depending on whether or not such operations have been designated, a split may be *serial*, or may be *independent*. In the former case, t_a must commit in order for t_b to commit, whereas in the latter, t_a and t_b can commit or abort independently.

After the split, t_a can split again creating another split transaction t_c. Split transactions can further split creating new split transactions. This leads to a different type of hierarchically structured transactions from those of nested transactions and distributed transactions. See Figure 4.3.

Let us further examine the semantics of the split transactions using the ACTA formalism. In the split transaction model, a transaction can be initiated through either the Begin event, called *primary* transaction, or the Split event, called *split* transaction. Although primary and split transactions are associated with different significant events [Axioms 1 and 4], their corresponding events share the same semantics.

$\mathsf{Split}_{t_a}[t_b, CanAccess]$ splits a primary or a split transaction t_a into a *splitting* transaction t_a and *split* transaction t_b. Since the idea is to allow the splitting transaction to give the split transaction the responsibility for finalizing some of its operations (these are the operations in the *DelegateSet*), the Split event is partially specified in terms of the delegation event $Delegate_{t_a}[t_b, DelegateSet]$ [Axiom 12]. Here, it is interesting to note that, in contrast to transactions initiated by the Begin event, through delegation, split transactions can affect objects in the database by committing or aborting delegated operations and without invoking any operation on them.

Further, the splitting transaction has the ability for allowing the split transaction to view some of its operations on some objects without conflict (these are the operations in the *CanAccess*) [Axiom 11]. However, the splitting transaction cannot view the operations of the split transaction on the same objects. A splitting transaction that allows the view of some of its operations on an object, say *ob*, can still invoke an operation on *ob* after the split as long as the split transaction has not invoked a conflicting operation on *ob* [Axiom 20].

A split is independent, if $CanAccess$ is empty. In the case of *serial split* in which $CanAccess$ is not empty, t_b develops an abort dependency on t_a[1] [Axiom 14].

DEFINITION 4.12: *Axiomatic definition of Split Transactions*

t_p denotes a primary transaction.

t_a denotes a primary or split transaction which splits.

t_b denotes the split transaction of t_a.

t denotes a transaction, primary or split.

1. $SE_{t_p} = \{\text{Begin, Split, Commit, Abort}\}$
2. $IE_{t_p} = \{\text{Begin}\}$
3. $TE_{t_p} = \{\text{Commit, Abort}\}$
4. $SE_{t_b} = \{\text{Split, Commit, Abort}\}$
5. $IE_{t_b} = \{\text{Split}\}$
6. $TE_{t_b} = \{\text{Commit, Abort}\}$
7. t satisfies the fundamental Axioms I to IV
8. $\forall ob\ \exists p\ (p_t[ob] \in H) \Rightarrow (ob\ is\ atomic)$
9. $View_t = H_{ct}$
10. $ConflictSet_{t_p} = \{p_t[ob] \mid t \neq t_p, \mathcal{I}nprogress(p_t[ob])\}$
11. $ConflictSet_{t_b} = \{p_t[ob] \mid (t \neq t_b, t \neq t_a, \mathcal{I}nprogress(p_t[ob])) \vee (t = t_a, \mathcal{I}nprogress(p_t[ob]) \wedge (p_t[ob] \notin CanAccess))\}$
12. $(\text{Split}_{t_a}[t_b, CanAccess] \in H) \Leftrightarrow (Delegate_{t_a}[t_b, DelegateSet] \in H)$
13. $(DelegateSet \subset AccessSet_{t_a}) \wedge (CanAccess \subset AccessSet_{t_a}) \wedge (DelegateSet \cap CanAccess = \phi)$
14. $post(\text{Split}_{t_a}[t_b, CanAccess]) \Rightarrow ((CanAccess \neq \phi) \Rightarrow ((t_b\ \mathcal{AD}\ t_a) \in DepSet_{ct}))$
15. $\exists ob\ \exists q\ \exists t_i\ (Commit_t[q_{t_i}[ob]] \in H) \Rightarrow (\text{Commit}_t \in H)$
16. $(\text{Commit}_t \in H) \Rightarrow \forall ob\ \forall q\ \forall t_i\ ((ResponsibleTr(q_{t_i}[ob]) = t) \Rightarrow (Commit_t[q_{t_i}[ob]] \in H))$

---

[1]By taking into consideration the semantics of operations on the individual objects in *CanAccess*, it would be possible to induce weaker dependencies, e.g. commit dependency, rather than abort dependency.

17. $\exists ob\ \exists q\ \exists t_i\ (Abort_t[q_{t_i}[ob]] \in H) \Rightarrow (\mathsf{Abort}_t \in H)$
18. $(\mathsf{Abort}_t \in H) \Rightarrow$
    $\forall ob\ \forall q\ \forall t_i\ ((ResponsibleTr(q_{t_i}[ob]) = t) \Rightarrow (Abort_t[q_{t_i}[ob]] \in H))$
19. $(\mathsf{Commit}_t \in H) \Rightarrow \neg(tC_N^*t)$
20. $\forall t \in SplitOf\,(t_a)\ \forall ob\ \forall q$
    $pre(q_{t_a}[ob]) \Rightarrow (\nexists p\ ((p_t[ob] \in H_{ct}) \wedge conflict(p_t[ob], q_{t_a}[ob]))$

SplitOf(t) is the set of transactions derived by a sequence of splits from the transaction t.

As in the case of nested transactions, Axioms 7, 8 and 19 are not sufficient to ensure serializability of split transactions due to Axioms 11 and 12. However, split transactions are serializable as we show below.

DEFINITION 4.13: A transaction t_i *behaves like* another transaction t_j if t_i has the same properties as t_i although t_i and t_j are associated with different significant events ($SE_{t_i} \neq SE_{t_i}$).

LEMMA 4.7: A primary transaction t_p *behaves like* an atomic transaction, if it does not split.

PROOF:
1. Axioms 7, 9, 10 and 8 of split transactions correspond to Axioms 4, 5, 6 and 7 of atomic transactions, respectively.
2. Axioms 19 and 15–18 of spit transaction are generalizations of Axioms 8 and 9–12 of atomic transactions.
3. If t_p does not split, Axioms 11–14 and 20 of split transactions do not apply.
4. By (1), (2) and (3), t_p that does not split, satisfies the axiomatic definition of atomic transactions [Definition 4.2] except Axiom 1. □

LEMMA 4.8: Let t_a be the splitting transaction and t_b be the split transaction. If $CanAccess = \phi$ (i.e., independent split), then t_a and t_b are *serializable* (in any order).

PROOF:

1. If $CanAccess = \phi$, Axiom 14 does not apply and thus, due to split, no dependency is induced between t_a and t_b that constrains their commitment.
2. Given Axiom 11, if $CanAccess = \phi$, no conflicts are ignored between operations in progress. Thus if t_a and t_b invoke conflicting operations, then $(t_a C_N t_b) \vee (t_b C_N t_a)$.
3. By 1, 2 and Axiom 19, t_a and t_b are serializable in any order or one of the two orders depending on whether or not they invoke conflicting operations. □

LEMMA 4.9: Let t_a be the splitting transaction and t_b be the split transaction. If $CanAccess \neq \phi$, then t_a and t_b *commit serially*, i.e.,

1. $(\mathsf{Commit}_{t_b} \in H) \Rightarrow (\mathsf{Commit}_{t_a} \rightarrow \mathsf{Commit}_{t_b})$
2. t_a and t_b are *serializable.*

That is, in the case of serial split, if both splitting and split transactions commit then the splitting transaction commits before the split transaction.

PROOF: The first clause follows from Axiom 14 and corresponds to a logical rewriting of abort dependency $\mathcal{AD}$ using the four fundamental axioms [Axiom 4]. The second clause follows from Axiom 8 which corresponds to the criterion of serializable executions which considers the presence of delegation [definition 4.8]. □

LEMMA 4.10: Let t_a be the splitting transaction and t_b be the split transaction. If $DelegateSet = \phi$, then t_a and t_b are *failure atomic.*

PROOF: Given Axioms 16-18 and effectively no delegation ($DelegateSet = \phi$), this is similar to Lemma 4.1. □

LEMMA 4.11: Let t_a be the splitting transaction and t_b be the split transaction. If $DelegateSet = \phi$, then t_a and t_b are *serializable.*

PROOF: This follows from lemma 4.8, if ($CanAccess = \phi$), or from lemma 4.9, if ($CanAccess \neq \phi$). □

THEOREM 4.4: A set of split transactions T is *serializable.*

PROOF: To prove this we show (1) that the transactions derived by a sequence of splits from a primary t_s are serializable with each other, (2) that they are serializable with respect to all other transactions. The former follows from lemmas 4.8 and 4.9 and the latter from the semantics of delegation and Axioms 8 and 19. □

DEFINITION 4.14: A split transaction management scheme is *correct* if it conforms to definition 4.12.

4.5.2 Joint Transactions

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

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

DEFINITION 4.15: *Axiomatic definition of Joint Transactions*

t_a denotes a joining transaction.

t_b denotes a joint transaction.

t denotes either a joining or a joint transaction.

1. SE_t = {Begin, Join, Commit, Abort}
2. IE_t = {Begin}
3. TE_t = {Join, Commit, Abort}
4. t satisfies the fundamental Axioms I to IV
5. $\forall ob\ \exists p\ (p_t[ob] \in H) \Rightarrow (ob\ is\ atomic)$
6. $View_t = H_{ct}$
7. $ConflictSet_t = \{p_{t'}[ob] \mid t' \neq t, Inprogress(p_{t'}[ob])\}$
8. $(\text{Commit}_t \in H) \Rightarrow \neg(tC_N^*t)$
9. $\exists ob\ \exists q\ \exists t_i\ (Commit_t[q_{t_i}[ob]] \in H) \Rightarrow (\text{Commit}_t \in H)$

10. $(\mathsf{Commit}_t \in H) \Rightarrow$
    $\forall ob\ \forall q\ \forall t_i\ ((ResponsibleTr(q_{t_i}[ob]) = t) \Rightarrow (Commit_t[q_{t_i}[ob]] \in H))$
11. $\exists ob\ \exists q\ \exists t_i\ (Abort_t[q_{t_i}[ob]] \in H) \Rightarrow (\mathsf{Abort}_t \in H)$
12. $(\mathsf{Abort}_t \in H) \Rightarrow$
    $\forall ob\ \forall q\ \forall t_i\ ((ResponsibleTr(q_{t_i}[ob]) = t) \Rightarrow (Abort_t[q_{t_i}[ob]] \in H))$
13. $(\mathsf{Join}_{t_a}[t_b] \in H) \Leftrightarrow (Delegate_{t_a}[t_b, AccessSet_{t_a}] \in H)$

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

Axiom 13 specifies that when Join occurs, the joining transaction's objects are delegated to the joint transaction. This means the joining transaction's effects are made permanent to the database only if the joint transaction commits. In this regard, a joining transaction behaves similar to child transaction of a nested transaction when child transaction commits: Note the similarity between Axiom 13 of joint transactions and Axiom 8 of nested transactions. Because of this, joint transactions can be said to be compatible with nested transactions in the sense that joint transactions can be used to join nested transactions into a single nested transaction, known as a supertransaction [81] by making their root transactions children of the supertransaction.

LEMMA 4.12: A transaction t in the joint transaction model *behaves like* an atomic transaction if t commits or aborts, i.e., it does not join any other transaction.

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

PROOF:

1. Axioms 4–7 of joint transactions correspond to Axioms 4-7 of atomic transactions.
2. Axioms 8–12 of joint transactions are generalizations of Axioms 8–12 of atomic transactions.

3. If t does not join, delegation does not occur, and hence, Axiom 13 is in applicable.
4. Thus, from (1), (2) and (3), t behaves like an atomic transaction. □

LEMMA 4.13: A transaction t in the joint transaction model is *quasi failure atomic.*

PROOF:

1. Given that a joint transaction that commits or aborts is failure atomic [Lemma 4.12], if t is such a joint transaction, t is also quasi failure atomic. Failure atomicity implies quasi failure atomicity.
2. Given (1), the quasi failure atomicity of joining transactions can be shown by induction similar to the proof of the quasi failure atomicity of nested subtransactions [Lemma 4.4]. □

THEOREM 4.5: A joining transaction t_a may *not* be serializable with respect to the joint transaction t_b.

PROOF: Assume operations p, q and r defined on object ob conflict. Without loss of generality, consider the case in which t_a has invoked p before t_b has invoked q, and t_a has invoked r after t_b has invoked q. Since ob is an atomic object [Axiom 5], if p and q conflict, $(t_a C_N t_b)$ and, if q and r conflict, $(t_b C_N t_a)$. Thus, $(t_a C_N t_a)$ and t_a and t_b are not serializable. □

COROLLARY 1: A joining transaction t_a is *serializable* with respect to the joint transaction t_b if $(Join_{t_a}[t_b] \in H) \Rightarrow ((t_a C_N t_b) \wedge \neg(t_b C_N t_a))$.

The set of axioms resulting from adding this as an axiom to the axioms of joint transactions produces the axiomatic definition of multi-coloured actions [88].

DEFINITION 4.16: A joint transaction management scheme is *correct* if it conforms to definition 4.15.

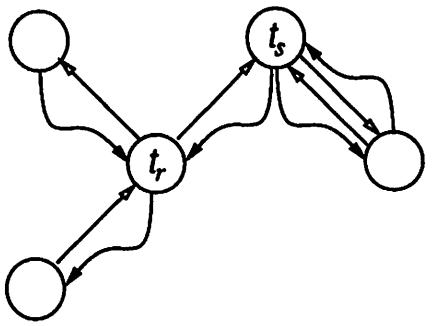

Figure 4.4 Structure of Recoverable Communicating Actions

4.6 Recoverable Communicating Actions

In the context of long and cooperative transactions, the *Recoverable Communicating Actions (RCA)* model has been proposed to deal with the problem of non hierarchical computations [97]. In this model, an action, the *sender*, is allowed to communicate with another action, the *receiver*, by sending results of operations, inducing an *abort dependency* of the receiver on the sender. If the sender aborts then the receiver must abort as a result of the dependency.

By developing abort dependencies, RCAs form a *recoverable computation*, a self-contained task or activity. For this reason, actions belonging to the same recoverable computation require synchronized commitment. That is, even in the case of a sender which has no abort dependencies on any other action, the sender cannot commit independently. However, partial failures are tolerated since an action t_r may abort without aborting the action t_s on which it has developed an abort dependency. In short, a recoverable computation can dynamically expand through the development of dependencies and shrink due to abortion of actions.

Here is the characterization of RCAs in ACTA:

DEFINITION 4.17: *Axiomatic definition of RCAs*

t_s denotes a sender.

t_r denotes a receiver.

t denotes either a sender or a receiver.

1. SE_t ={ Begin, Send, Receive, Commit, Abort}

2. IE_t ={Begin}
3. TE_t ={Commit, Abort}
4. t satisfies the fundamental Axioms I to IV
5. $\forall ob\ \forall p\ (p_t[ob] \in H) \Rightarrow (ob\ is\ atomic)$
6. $View_t = H_{ct}$
7. $ConflictSet_t = \{p_{t_i}[ob] | t_i \neq t, Inprogress(p_{t_i}[ob])) \wedge$
   $\forall j\ (p_{t_i}[ob] \notin Message_j(t_i, t))\}$
   where $Message_j(t_i, t)$ denotes the jth message of t_i to t
8. $\exists ob\ \exists p\ (Commit_t[p_t[ob]] \in H) \Rightarrow (\mathsf{Commit}_t \in H)$
9. $(\mathsf{Commit}_t \in H) \Rightarrow \forall ob\ \forall p\ ((p_t[ob] \in H) \Rightarrow (Commit_t[p_t[ob]] \in H))$
10. $\exists ob\ \exists p\ (Abort_t[p_t[ob]] \in H) \Rightarrow (\mathsf{Abort}_t \in H)$
11. $(\mathsf{Abort}_t \in H) \Rightarrow \forall ob\ \forall p\ ((p_t[ob] \in H) \Rightarrow (Abort_t[p_t[ob]] \in H))$
12. $(\mathsf{Commit}_t \in H) \Rightarrow \neg(tC^*t)$
13. $Message_i(t_s, t_r) \subseteq AccessSet_{t_s}$
14. $pre(\mathsf{Receive}_{t_r}[t_s, Message_i(t_s, t_r)]) \Rightarrow ((\mathsf{Send}_{t_s}[t_r, Message_i(t_s, t_r)] \in H_{ct}) \wedge$
    $(\mathsf{Receive}_{t_r}[t_s, Message_i(t_s, t_r)] \notin H_{ct}))$
15. $post(\mathsf{Receive}_{t_r}[t_s, Message_i(t_s, T_r)]) \Rightarrow (((t_r\ \mathcal{AD}\ t_s) \in DepSet_{ct}) \wedge$
    $((t_s\ \mathcal{CD}\ t_r) \in DepSet_{ct}))$

Axioms 1–3 state that Send and Receive are significant events for RCAs but they are neither initiation nor termination events. The specification of the other events, Begin, Commit and Abort are similar to those of atomic transactions. It is not hard to show that RCAs are failure atomic [Axioms 8–11 and no delegation occurs during the execution of RCAs] and serializable with respect to non-related RCAs [Axioms 5, 7 and 12].

The $\mathsf{Send}_{t_s}[t_r, Message_i(t_s, t_r)]$ event denotes the sending of ith message of transaction t_s to transaction t_r. $\mathsf{Receive}_{t_r}[t_s, Message_i(t_s, t_r)]$ corresponds to the receiving of the ith message of t_s by t_r. Every $Message_i(t_s, t_r)$ is a set of operations whose results t_s sends to t_r. By excluding the operations in a message from the t_r's conflict set, t_r cas effectively see the results of the sent operations without conflicts. Axiom 14 requires that for a Receive to occur, a corresponding Send should occur first, and states that a send corresponds to a single Receive.

Axiom 15 captures the composition relationship of a recoverable computation. The pair of commit and abort dependencies between sender and receiver guarantees the required synchronized commitment of the sender and receiver actions. Compare this with the pair of dependencies capturing the parent/child relationship in nested transactions: The abort dependency $\mathcal{AD}$ in RCAs corresponds to the weak-abort dependency $\mathcal{WD}$ in nested transactions. In spite of this, RCAs tolerate certain partial failures. A transactions t_r which has developed an abort dependency on another transaction t_s, can abort without aborting t_s. Furthermore, Axiom 7 of RCAs is a restricted form of Axiom 16 of nested transactions since Axiom 7 states that an RCA can ignore conflicts only with respect to *particular* operations (not any operation as in nested transactions) invoked by the sender ("its parent" and not by any of its ancestors as in nested transactions).

The abort dependency guarantees that the effects of aborted actions are not reflected in the database. Neither the abort nor the commit dependencies prevent an action from developing any new dependencies. It is even possible for an action to be both a sender and a receiver at the same time. In this manner, RCAs can produce non-hierarchical structures (see figure 4.4).

DEFINITION 4.18: A recoverable communicating action management scheme is *correct* if it conforms to definition 4.17.

4.7 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 properties. However, their 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.

DEFINITION 4.19: 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. $Views_S = \phi$
9. $View_t = H_{ct}$
10. $ConflictSet_S = \phi$
11. $ConflictSet_t = \{p_{t'}[ob] \mid t' \neq t, Inprogress(p_{t'}[ob])\}$
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. $(\text{Commit}_t \in H) \Rightarrow \neg(tC^*t)$
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. $post(\mathsf{Commit}_{T_i}) \Rightarrow (((CT_i\ \mathcal{CMD}\ S) \in DepSet_{ct}) \wedge$
$((CT_i\ \mathcal{BAD}\ S) \in DepSet_{ct}))$
where $1 \leq i < n$
21. $post(\mathsf{Begin}_{T_n}) \Rightarrow ((S\ \mathcal{SCD}\ T_n) \in DepSet_{ct})$

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[2] — this is the reason for the presence of a saga node, e.g., in Figure 4.5.

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

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

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 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.

Figures 4.5–4.9 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 4.14: 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.

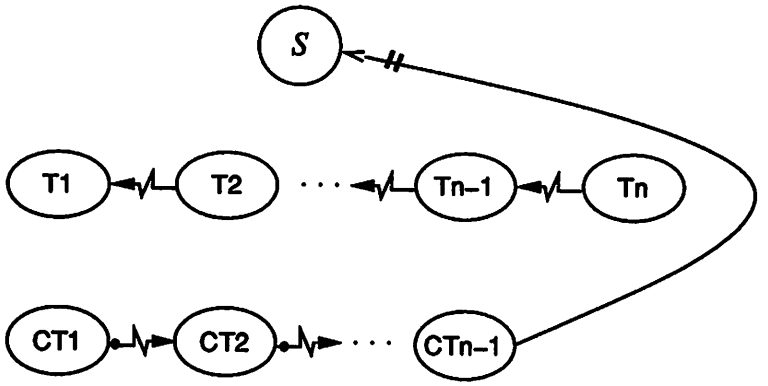

Figure 4.5 Structure of a just initiated Saga

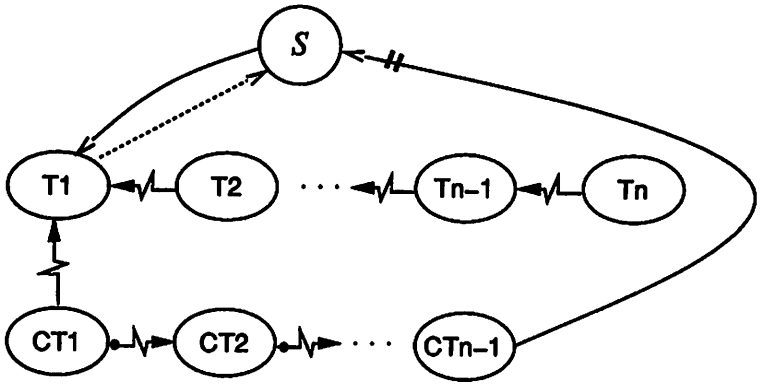

Figure 4.6 Structure of a Saga when component T_1 in progress

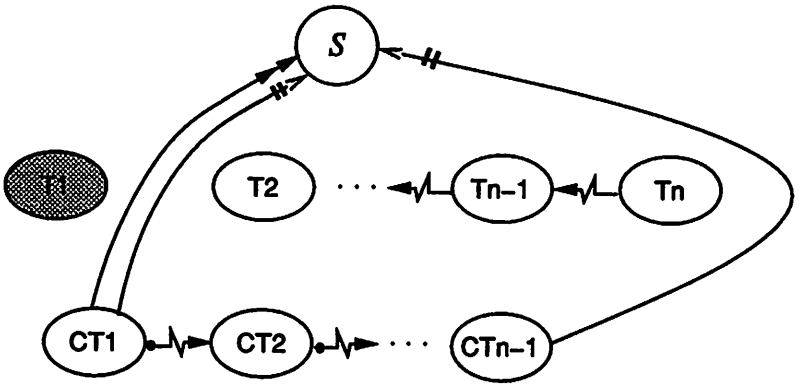

Figure 4.7 Structure of a Saga after component T_1 commits

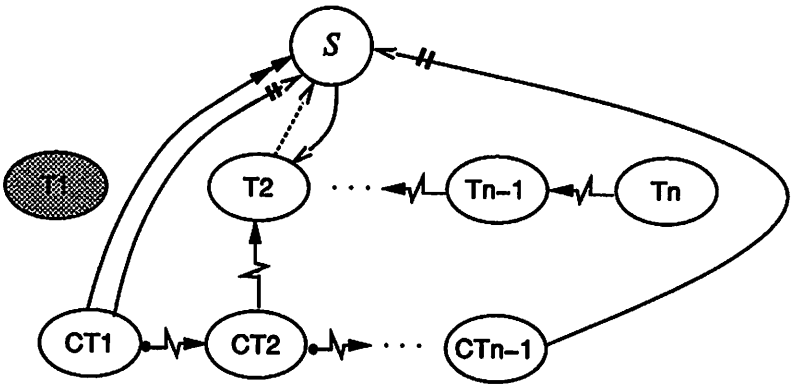

Figure 4.8 Structure of a Saga when component T_2 in progress

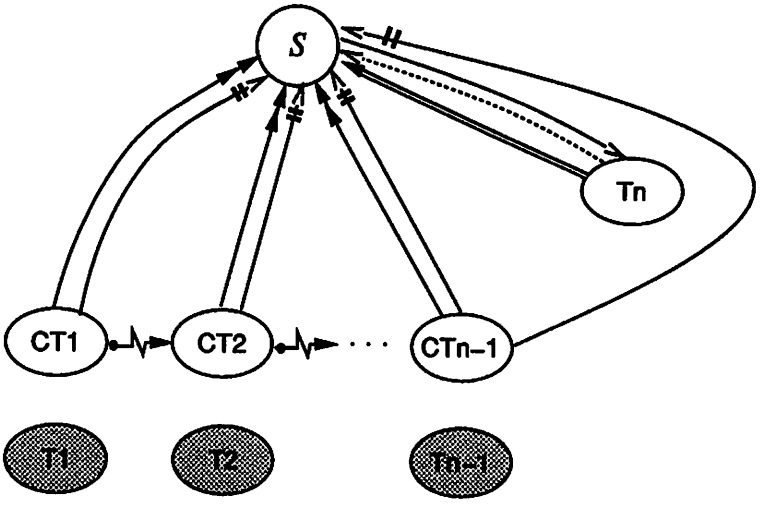

Figure 4.9 Structure of a Saga when component T_n in progress

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:

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

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 4.15: 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 4.14):

   $$(\mathsf{Commit}_{CT_1} \in H) \Rightarrow (\mathsf{Commit}_{T_1} \rightarrow \mathsf{Commit}_{CT_1})$$

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 4.6: 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 4.14 and 4.15 and the committed projection of the history.

4.7.1 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 4.20: t denotes either a T_i, a component transaction, or a CT_i, a compensating transaction of T_i.

1. $SE_t = \{\text{Begin, Commit, Abort}\}$
2. $IE_t = \{\text{Begin}\}$
3. $TE_t = \{\text{Commit, Abort}\}$
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. $\exists ob\ \exists p\ (Commit_t[p_t[ob]] \in H) \Rightarrow (\text{Commit}_t \in H)$
8. $(\text{Commit}_t \in H) \Rightarrow \forall ob\ \forall p\ ((p_t[ob] \in H) \Rightarrow (Commit_t[p_t[ob]] \in H))$
9. $\exists ob\ \exists p\ (Abort_t[p_t[ob]] \in H) \Rightarrow (\text{Abort}_t \in H)$
10. $(\text{Abort}_t \in H) \Rightarrow \forall ob\ \forall p\ ((p_t[ob] \in H) \Rightarrow (Abort_t[p_t[ob]] \in H))$
11. $\forall ob\ \exists p\ (p_t[ob] \in H) \Rightarrow (ob\ is\ atomic)$
12. $(\text{Commit}_t \in H) \Rightarrow \neg(tC^*t)$
13. $post(\text{Begin}_{T_i}) \Rightarrow (((T_i\ \mathcal{BCD}\ T_{i-1}) \in DepSet_{ct}) \wedge$
    $((CT_j\ \mathcal{WCD}\ CT_{j+1}) \in DepSet_{ct})) \wedge$
    $((CT_{n-1}\ \mathcal{BAD}\ T_n) \in DepSet_{ct})$
    where $1 < i \leq n$ and $1 \leq j < n-1$
14. $post(\text{Begin}_{T_i}) \Rightarrow ((CT_i\ \mathcal{BCD}\ T_i) \in DepSet_{ct})$, where $1 \leq i < n$
15. $post(\text{Commit}_{T_i}) \Rightarrow (((CT_i\ \mathcal{CMD}\ T_{i+1}) \in DepSet_{ct}) \wedge$
    $((CT_i\ \mathcal{CMD}\ T_n) \in DepSet_{ct})$
    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.

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 4.10 and 4.11 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 4.14 and 4.15 and theorem 4.6 hold for the special saga.

4.8 Conclusion

While Chapter 3 introduced the formalism underlying ACTA, the expressive power of ACTA was portrayed in this chapter by formally specifying and reasoning about the properties of six popular extended transaction models. For each model, an axiomatic definition was given. Some of the axioms specify invariant assertion about the histories generated by the transactions of the particular model. Others provide explicit precondition or postcondition for an operations and transaction management primitives. The definition of the view and the conflict set associated with each transaction are also part of the axiomatic definition. Based on these axiomatic definitions, the failure semantics and consistency properties, for example, of distributed transactions, nested transactions and split transactions were derived. In addition, the axiomatic definitions provided the basis for comparisons between extended transactions models, e.g., distributed transactions and nested transactions.

The atomic transaction model was included among these models for two reasons. First, atomic transactions constitute the base model and form the basis for many of the extended transaction models. Second, we could demonstrate that while the ACTA formalism captures the results of the classical serializability theory [17, 78], it models serializability and failure atomicity down to the objects level. By supporting the modeling at a lower level of abstraction, ACTA can express object specific properties in a straightforward manner. It also provides the foundations for capturing other correctness criteria of extended transactions such as quasi failure atomicity and cooperative serializability.

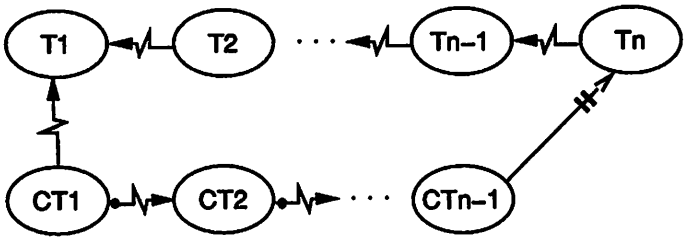

Figure 4.10 Structure of a special Saga before component T_1 commits

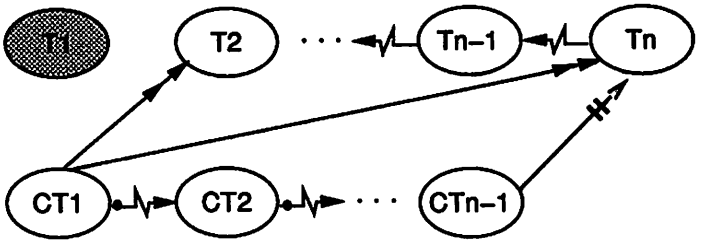

Figure 4.11 Structure of a special Saga after component T_1 commits

CHAPTER 5

SYNTHESIS OF EXTENDED TRANSACTION MODELS

The previous two chapters introduced the formalism underlying ACTA and demonstrated its expressive power by using it to define extended transaction models in an axiomatic form, specify correctness properties of the models, and prove that a particular model satisfies the specified properties. This chapter focuses on another aspect of ACTA, its use to derive new extended transaction models, which also illustrates the power of ACTA. That is, this chapter presents *ACTA as a tool for the synthesis of extended transaction models*, one which supports the development and analysis of new extended transaction models in a systematic manner.

New transaction definitions can be derived by slightly modifying the specifications of existing transaction models, or combining their specifications. Examples of the former include *Chain Transactions* (Section 5.1.1), *Reporting transactions* (Section 5.1.2) and *Co-Transactions* (Section 5.1.3) which are variations of joint transactions. As an example of the latter, the *nested-split* transaction model is derived in Section 5.2 from the specifications of nested and split transaction models. Section 5.3 presents three variations of Sagas which exhibit different degrees of flexibility.

5.1 Variations of Joint Transactions

In this section, we derive three new extended transaction models, namely, *chain transactions, reporting transactions* and *co-transactions*, though a series of manipulations, beginning with the axiomatic definition of joint transactions [Definition 4.15]. In [27], we defined these models using *dependency production rules*, a formalism close to dependency graphs which capture the static structure and the dynamics

of the evolution of the structure of transactions. Here we use axiomatic definitions to express the properties of these transaction models.

Notation: Throughout this section, we use t_a to denote a joining transaction, and t_b to denote a joint transaction.

5.1.1 Chain Transactions

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

DEFINITION 5.1: *Axiomatic definition of Chain Transactions*

t_a denotes a joining transaction.

t_b denotes a joint transaction.

t denotes either a joining or a joint transaction.

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

12. $(\mathsf{Abort}_t \in H) \Rightarrow$
    $\forall ob\ \forall q\ \forall t_i\ ((ResponsibleTr(q_{t_i}[ob]) = t) \Rightarrow (Abort_t[q_{t_i}[ob]] \in H))$
13. $(\mathsf{Join}_{t_a}[t_b] \in H) \Leftrightarrow (Delegate_{t_a}[t_b, AccessSet_{t_a}] \in H)$
14. $(Join_{t_a}[t_b] \in H) \Rightarrow \nexists t, (Join_t[t_b] \rightarrow Join_{t_a}[t_b])$

All the lemmas and theorems expressing the correctness properties of joint transactions (Section 4.5.2) hold also for chain transactions.

5.1.2 Reporting Transactions

A variation of the Joint Transaction model is the transaction model in which Join is not a termination event (Join $\notin TE_t$). A joining transaction continues its execution and periodically *reports* its results to the joint transaction by delegating more operations to the joint transaction. We call these transactions *Reporting Transactions.* Reporting transactions must invoke either Commit or Abort to complete their computation [Axiom 3].

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

DEFINITION 5.2: *Axiomatic definition of Reporting Transactions*

t_a denotes a joining transaction.

t_b denotes a joint transaction.

t denotes either a joining or a joint transaction.

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

10. $(\mathsf{Commit}_t \in H) \Rightarrow$
    $\forall ob\ \forall q\ \forall t_i\ ((ResponsibleTr(q_{t_i}[ob]) = t) \Rightarrow (Commit_t[q_{t_i}[ob]] \in H))$
11. $\exists ob\ \exists q\ \exists t_i\ (Abort_t[q_{t_i}[ob]] \in H) \Rightarrow (\mathsf{Abort}_t \in H)$
12. $(\mathsf{Abort}_t \in H) \Rightarrow$
    $\forall ob\ \forall q\ \forall t_i\ ((ResponsibleTr(q_{t_i}[ob]) = t) \Rightarrow (Abort_t[q_{t_i}[ob]] \in H))$
13. $(\mathsf{Join}_{t_a}[t_b] \in H) \Leftrightarrow (Delegate_{t_a}[t_b, AccessSet_{t_a}] \in H)$
14. $post(Join_{t_a}[t_b]) \Rightarrow ((t_a\ \mathcal{AD}\ t_b))$
15. $(Join_{t_a}[t_b] \in H) \Rightarrow \nexists t, t \neq t_a, (Join_{t_a}[t] \rightarrow Join_{t_a}[t_b])$
16. $(Join_{t_a}[t_b] \in H) \Rightarrow (Join_{t_b}[t_a] \notin H)$

The abort-dependency induced by Axiom 14 effectively maintains the termination semantics of joining transactions in the joint transaction model by guaranteeing the abortion of the joining transaction t_a when the joint transaction t_b aborts. Because Axiom 15 prevents t_a from joining a third transaction, t_a cannot report to any transaction other than t_b. Furthermore, Axiom 16 prevents t_b from joining back t_a. It is possible to allowing some of the operations on objects to be delegated with Join by replacing $AccessSet_{t_a}$ with a $ReportSet_{t_a} \subseteq AccessSet_{t_a}$ in this axiom 13.

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

5.1.3 Co-Transactions

The characterization of reporting transactions allows t_a to continue its execution but prevents t_b from joining t_a [Axiom 15]. Suppose t_a is suspended when it joins t_b and also t_b is allowed to join t_a. The transaction t_a can be suspended, if, at the join, its View Set becomes empty. We call this *view curtailment.* t_a is effectively suspended since after t_a delegates all the operations performed by t_a (in its Access Set) to t_b. Due to view curtailment, t_a can no longer access any object in the system. t_a will be able to resume execution when t_b joins t_a. This is because after the join t_a's view will be restored while t_b's is curtailed. We call these transactions

co-transactions because they behave like *co-routines* in which control is passed from one transaction to the other transaction at the time of the delegation and they resume execution where they were previously suspended. In the co-transaction model specified below, the View Set of the co-transaction that resumes execution is restored to H_{ct}.

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

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

DEFINITION 5.3: *Axiomatic definition of Co-Transactions*

t_a denotes a joining transaction.

t_b denotes a joint transaction.

t denotes either a joining or a joint transaction.

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

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

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

5.2 Nested-Split Transactions

Given our definitions for atomic transactions (see Definition 4.2), nested transactions (see Definition 4.10) and split transactions (see Definition 4.12) in axiomatic form, it is not difficult to see which axioms reflect the differences between these models and which axioms capture the similarity between them.

For instance, the Begin, Abort, and Commit events in the split transaction model have the same semantics as those for the *root transactions* in the nested transaction model (which are the same as those of atomic transactions). However, although at first glance the Spawn event in nested transactions and the Split event in split transactions appear to have similar semantics, their precise definitions show the actual differences in the induced dependencies. Specifically, whereas the Spawn event induces a commit dependency and a weak-abort dependency between the spawning and the spawned transactions [Axiom 17], the Split event induces an abort dependency of the split transaction on the splitting transaction [Axiom 14]. In addition, in contrast to the Spawn event, due to delegation, the Split event may associate a non-empty access set with the split transaction. Turning to similarities, it is possible to prove that both nested and split transactions produce only hierarchical transaction structures.

Given the similarities and differences between two models, the question of whether the two transaction models can be used in conjunction becomes important. Let us consider combining aspects from the nested and split transaction models. We would like to check whether the resulting model, called the *nested-split transaction model*, retains the properties of the two original models.

We would like to point out that the analysis below makes use of a pictorial representation of the relevant axioms, in contrast to the logical, i.e., axiomatic,

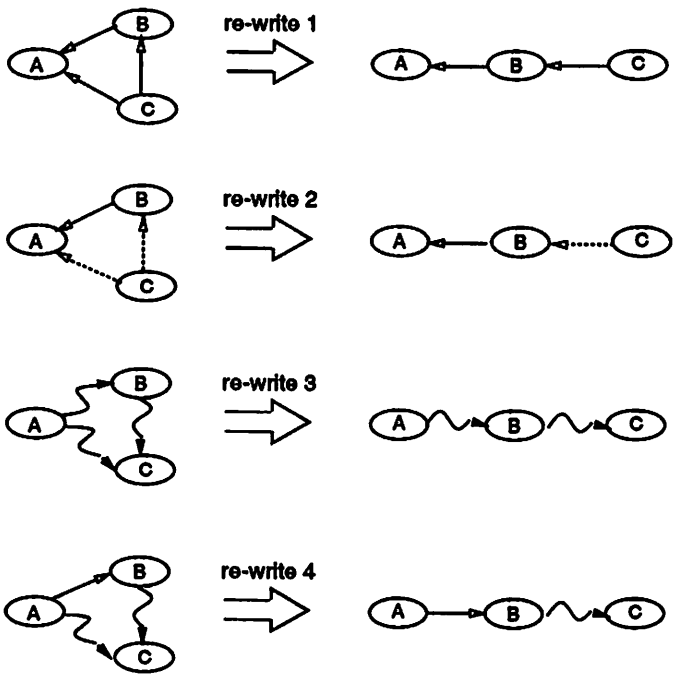

Figure 5.1 Semantics-preserving Re-Write Rules

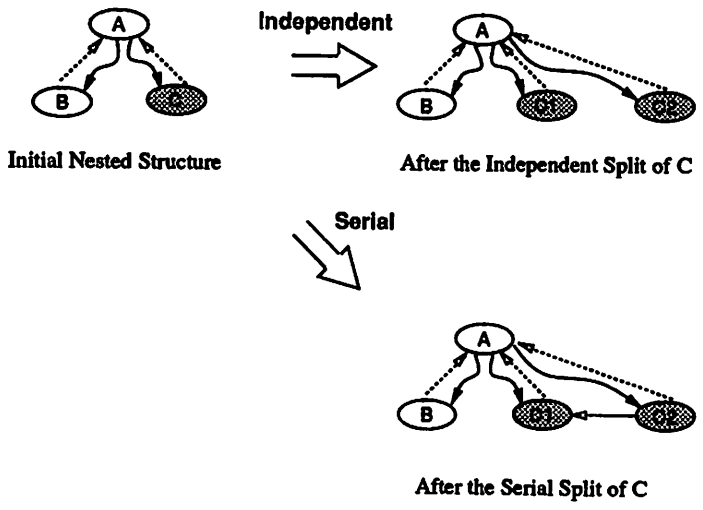

Figure 5.2 Splitting a Leaf Nested Subtransaction

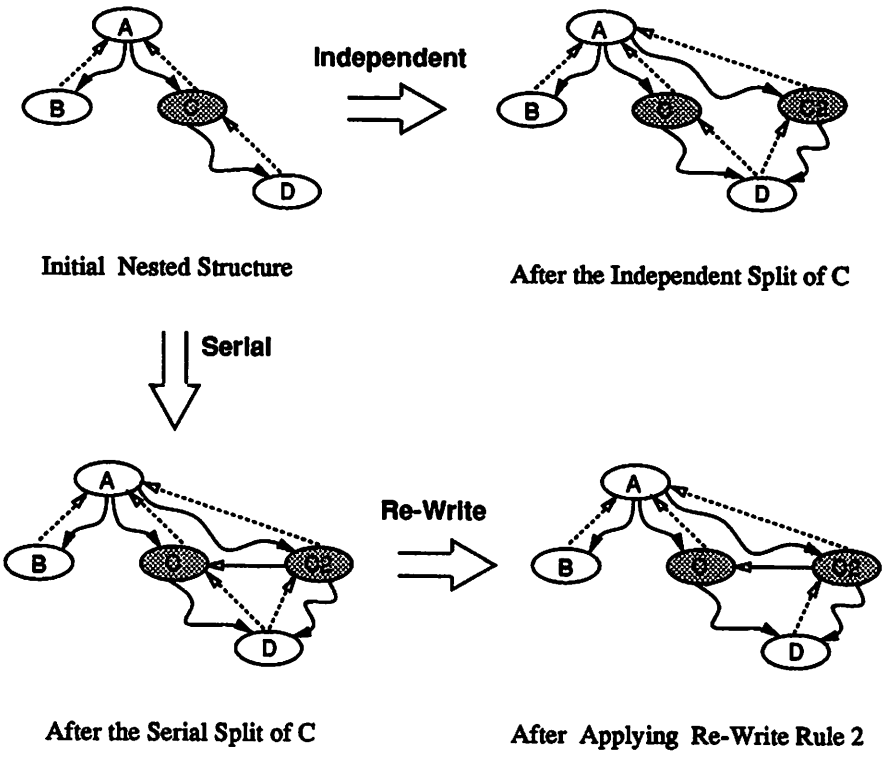

Figure 5.3 Splitting an Internal Nested Subtransaction

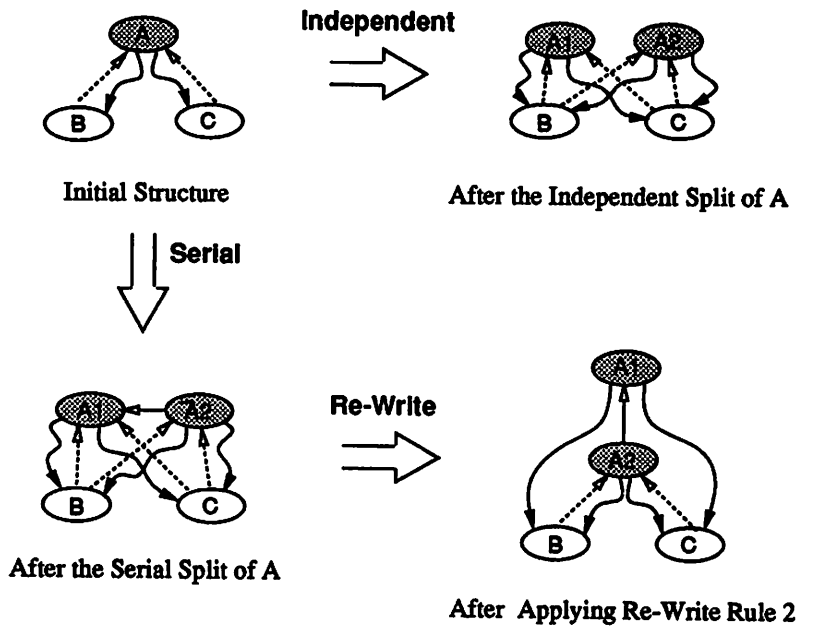

Figure 5.4 Splitting a Root Transaction

representation used in the previous section. It is not hard to see that the two, while having similar semantic properties, have different aesthetic properties.

Note that, given a nested transaction, it is possible to split a leaf node, an internal node, or a root node. The split nodes could execute independently or serially. Figures 5.2–5.4 capture the effects for all possible combinations. The dependencies shown follow from the specifications of dependencies for nested transactions [Axiom 17] and split transactions [Axiom 14]. (In these figures a *dotted arrow* denotes a weak-abort-dependency, a *solid arrow* denotes an abort-dependency, and a *squiggly arrow* denotes a commit-dependency.)

When a child node, say C (Figure 5.2), splits into two subtransactions, say $C1$ and $C2$, where $C1$ is the original subtransaction C, the dependencies between subtransaction C and its parent, transaction A, are assumed to hold between $C2$ and A. Since split transactions involve abort-dependencies, a node splitting may result in a subtransaction that has both abort-dependency and weak-abort-dependency on two other subtransactions (Figure 5.2, After the Serial Split of C).

Consistency preserving re-write rules are used to simplify the structure of an extended transaction by eliminating redundant dependencies. Figure 5.1 shows four such rewrite rules of which *re-write 2* simplifies the structure after a serial split of an internal node or a root node. The correctness of these rules can be formally established by invoking the semantics of the dependencies.

After applying the rewrite rules (in these cases only re-write 2 is applicable), we examine the remaining dependencies for each type of nested-split transaction to see if the resulting structure preserves the semantics of the nested and split transactions models. We conclude that only in the two cases where a leaf node is split are the properties of the two models preserved. In the case involving the splitting of a leaf node into two independent subtransactions (i.e., when $CanAccess = \phi$), the dependencies that the split transaction has on A are the same as those of the splitting transaction. In the case involving the serial splitting of a leaf node, an additional abort dependency is developed between the split transaction and the splitting transaction. But this is not disallowed by the nested transaction model. In fact, such abort dependencies may develop when children have conflicts over objects that they share.

In all other cases the model either establishes dependencies which destroy the hierarchical structure of the nested transactions or eliminates some of the dependencies required by the nested transactions. For example, in Figure 5.3, in the case of the independent split of C, subtransaction D has weak-abort-dependencies on two ancestors, C and $C2$ which clearly is disallowed by the hierarchical structure of nested transaction model.

Based on the above analysis we can state that properties of both nested and split transactions can be retained as long as only the splitting of leaves are allowed.

Even if splitting nodes is restricted to the splitting of leaf nodes, nested-split transactions are a useful new transaction model in a cooperative environment. Observe that an internal node becomes a leaf node any time that it has no active child subtransactions. That is, in nested-split transactions, a node may split at any point after all its child subtransactions have terminated and before activating any new subtransactions. For example, when subtransaction D terminates the structure that results is similar to that of Figure 5.2 (Initial nested Structure), node C can be split into two independent subtransactions $C1$ and $C2$ as in Figure 5.2 (After the Independent Split of C). $C1$ may continue the execution of C spawning new subtransactions, while $C2$ may commit delegating its objects to A. Since conflicts relative to the operations performed by A are not considered by the descendants of A (due to conflict set specifications of nested transactions), the operations delegated to A by $C2$ do not conflict with operations invoked by B. This effectively achieves cooperation between the original siblings C and B while they are still executing. In nested transactions, two siblings cannot cooperate while both siblings are active, since subtransactions delegate their objects to their parent only at commit time. Thus, nested-split transactions support a higher level of visibility between subtransactions than nested transactions. (A similar type of interaction occurs in the extended nested transaction model proposed in [71].)

This study shows the efficacy of the ACTA model in determining the properties of new transaction models, in this case, one derived by combining existing models.

5.3 Flexible Sagas

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.

5.3.1 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 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 5.5 shows the structure of such a saga resulting from the modified Axiom 18. This corresponds to Figure 4.5 that represents the structure of the original saga [Definition 4.19].

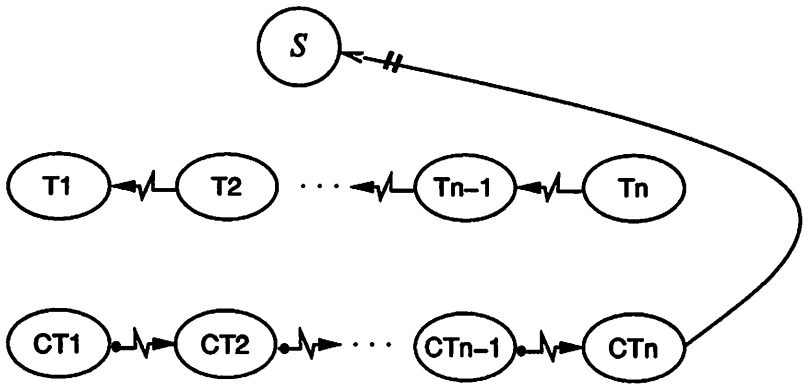

Figure 5.5 Structure of a Saga without special relation with T_n

5.3.2 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 5.6). 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 5.7. 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 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

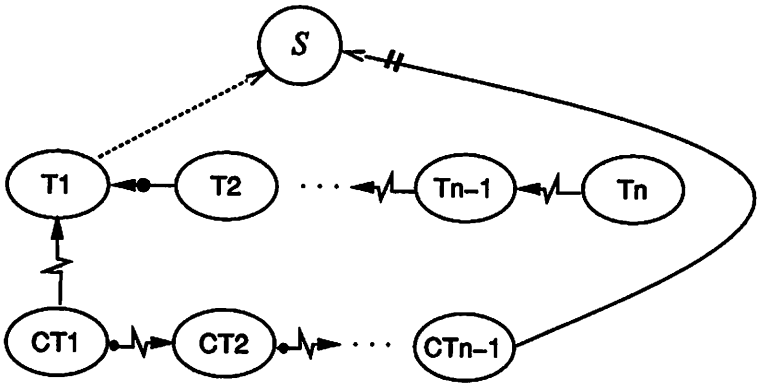

Figure 5.6 Structure of a Saga when non-vital component T_1 in progress

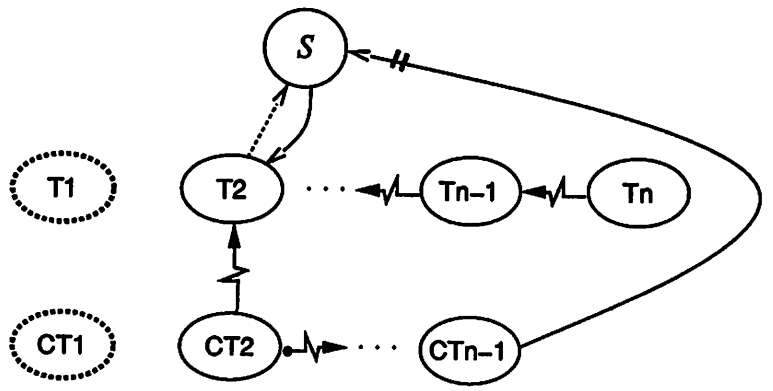

Figure 5.7 Structure of a Saga after non-vital component T_1 aborts

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

and corresponds to the axiomatic definition of the transaction model proposed in [21, 44].

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 4.6.

THEOREM 5.1: 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:
$$\forall i, 2 < i \leq n \ (\mathsf{Commit}_{T_{i-1}} \rightarrow \mathsf{Commit}_{T_i})$$
3. For all committed components, their compensating transactions commit in the required order:
$$\exists k, 1 \leq k \leq n \ \forall i, 1 < i < k \ ((\mathsf{Commit}_{T_{i-1}} \rightarrow \mathsf{Commit}_{T_i}) \wedge$$
$$(\mathsf{Commit}_{T_k} \rightarrow \mathsf{Commit}_{CT_k}) \wedge (\mathsf{Commit}_{CT_i} \rightarrow \mathsf{Commit}_{CT_{i-1}}))$$

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 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 [21, 37] and contingency transactions [21] 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[1] which are component transactions of sagas (Figures 5.8 and 5.9).

5.3.3 Sagas of Sagas

Dependencies between a nested saga transaction and the components of the nested saga are different from those of a (top) saga transaction on its 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 5.8, 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

[1]Nested Sagas corresponds to a class of sagas with complex structure and hence, it is different from the nested saga model proposed in [44].

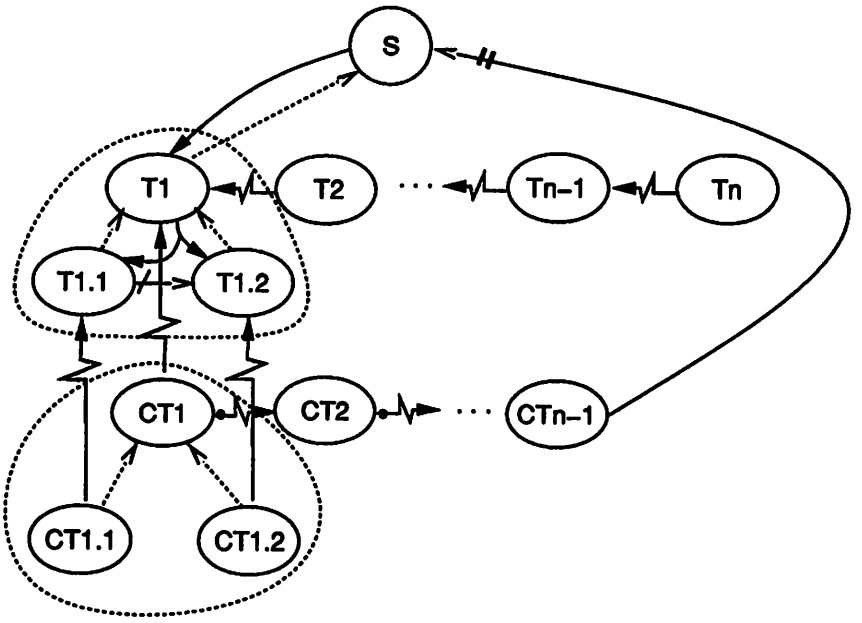

Figure 5.8 Saga structure when nested saga component T_1 in-progress

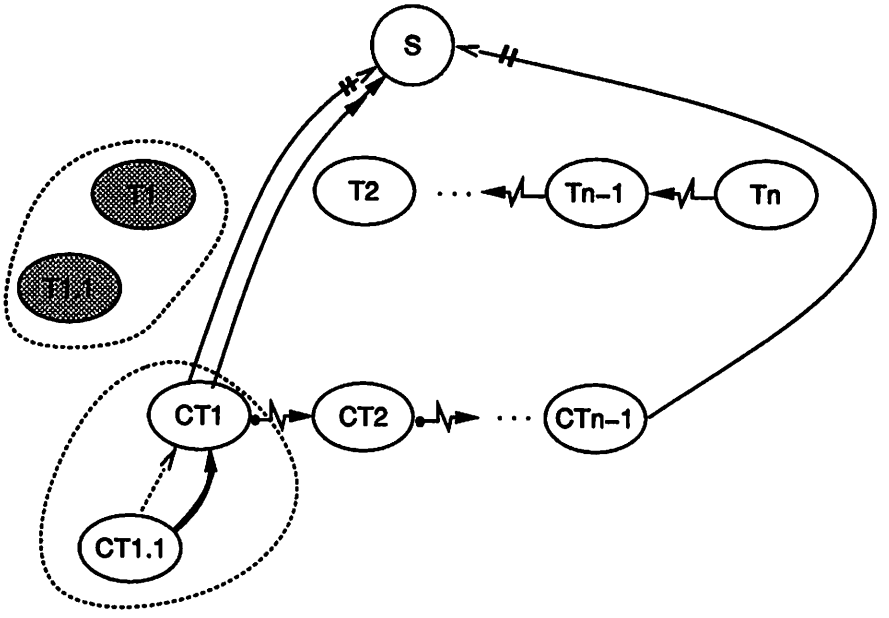

Figure 5.9 Saga structure after nested saga component T_1 commits

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 5.8, 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 5.8) 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.

5.3.4 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.

5.4 Conclusion

This chapter completes the presentation of ACTA by portraying how new extended transaction models can be developed using the ACTA formalism. With specific properties in mind, seven such extended transaction models are derived from the specifications of existing ones. Specifically, chain transactions were a result of a restriction imposed on the invocation of the Join event associated with joint transactions such that they result in linear structures only. This restriction was captured by an axiom which when added to the axiomatic definition of joint transactions yields the definition of chain transactions. Also, reporting transactions and co-transactions were derived from joint transactions by removing the restriction that Join be a terminating event. This allows a transaction to join multiple times

with another transaction, thereby delegating more operations to the joint transaction. Co-transactions are more flexible than reporting transactions since they allow transactions to join back and forth.

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

To derive a more flexible saga model in which failed components can be retried, replaced with alternative ones, or ignored, new saga definitions were proposed and showed to be mutations of the original definition. More flexibility was achieved by introducing new component transaction types, new significant events associated with these types, and new dependencies describing the relationship between these new transaction types.

These new extended transactions models show that besides supporting the specification and analysis of existing transaction models, ACTA has the power to support the synthesis and analysis of new extended transaction models in a systematic way, as well as the customization of existing ones. Thus, handling the requirements of new applications is facilitated.

CHAPTER 6

SUMMARY AND FUTURE WORK

6.1 Summary

This dissertation has proposed the ACTA framework, a novel approach to formally specify and reason about extended transaction models. As summarized below, the ACTA formalism allows for a precise definition of the properties possessed by transactions in a particular model, *vis a vis* visibility, consistency, recovery and permanence.

In ACTA, the visibility properties of a transaction are specified in terms of its *view* and *conflict set*. Specifically, a transaction's view specifies the state of objects visible to that transaction while the transaction's conflict set contains those operations against which conflicts have to be determined.

ACTA permits the specification of the semantics of significant events pertaining to a transaction model by stating how the invocation of a significant event by one transaction affects those invoked by another. Also, the effects of transactions on each other when they access shared objects is specified through the conflict relationships between operations. Consistency related correctness criteria then specify which effects are allowable, which are not, and which are required. For example, consistency in atomic transactions is equated with serializability and we saw in Chapter 3 how this is captured in ACTA. In the same chapter, we saw how other correctness properties are captured in ACTA, e.g., predicatewise serializability, cooperative serializability, and objectwise serializability, i.e., the serializability property of individual objects. Formally expressing correctness properties with ACTA, allows for a formal proof that a particular transaction model satisfies the specified correctness properties. In Chapter 4, we gave such proofs for distributed transactions, nested transactions, split transactions, and sagas.

Recovery properties of objects are specified in ACTA though dependencies induced when operations are executed on the object (See definition of a *correct* object in Chapter 3). Further, the specification of the "abort" event indicates the explicit effects of the abort of a transaction. For instance, Axiom 17 for nested transactions state that the abortion of a child transaction forces the abortion of all its operations. The *no orphan commits* lemma says that a child transaction cannot commit after its parent has committed or aborted. This lemma follows directly from the dependencies that arise from the structural properties of nested transactions. Finally, delegation supports the specification of the recovery properties of objects either directly through the specification of the effect of an "abort" event or indirectly through the specification of other significant events. For an example of the latter, consider Axiom 12 for split transactions which allows the splitting transaction to relinquish some of the objects in its access set to the split transaction. This binds the recovery of these objects with the split transaction's recovery properties.

ACTA captures permanence by explicitly modeling (and separating) the commitment of operations and the commitment of transactions. The former make the changes made by an operation permanent. In certain models, e.g., atomic transactions, commitment of a transaction occurs if and only if all the operations can be committed, i.e., made permanent. On the other hand, in the nested transaction model, a child transaction commits without making the effects of its operations permanent; these effects are delegated to its parent.

It is worth noting that ACTA does not make any assumptions about the structure of the transactions; for example, it does not assume a hierarchical structure as in transaction groups [89]. It is flexible enough to allow the definition of new dependencies as new significant events are associated with extended transactions. These suggest that it is a general framework.

Finally, it should be noted that ACTA is applicable to different transaction management mechanism, conflict resolution mechanism, or recovery mechanism. For example, given the formal definition of split transactions, a correct implementation can be realized using, e.g., (1) cooperative locks [41, 42, 61] to control the visibility between a splitting transaction t_a and a split transaction t_b to the objects in $CanAccess$ while preventing any other transaction t_c from accessing these objects,

or by (2) aborting any transaction t_c that is inconsistent with the *commit serially* property. (Note regarding (1) that traditional locks can support only independent split transactions by changing the lock holder's identifier for objects in *DelegateSet*).

An issue that concerns specifications in general is whether all "aspects of interest" have been specified. While this depends very much on whether a specifier has been able to "cover" all the aspects, the technique used for the specification should prompt the specifier to consider all aspects of interest. We believe that ACTA has this characteristic. First of all, it demands that all significant events of interest be identified and then expects specifications for the effects of each event on other events as well as on the visibility of objects via view and conflict set changes and delegation. The correctness properties expected of a transaction model can be specified and if the attempt at proving them, given the axiomatic definition of the model, fails then it is an indication that either the axioms are insufficient or the correctness properties do not hold.

Besides facilitating analysis of individual transaction models, ACTA provides a formalism that unifies existing transaction frameworks making it possible to ascertain the similarities and differences between two models. As we saw in Chapter 4, given our definitions for atomic transactions, distributed, nested transactions, split transactions, and sagas in axiomatic form, it was not difficult to see which axioms reflect the differences between these models and which axioms capture the similarity between them.

Given the similarities and differences between two models, the question of whether two transaction models can be used in conjunction becomes important. Once, again, using the formalism, and in particular, correctness-preserving transformation rules, we showed in Chapter 5 that properties of both nested and split transactions can be retained as long as only the splitting of leaves are allowed. This also illustrated that ACTA can be used as a tool for synthesis of extended transaction models, one which supports the development and analysis of new extended transaction models in a systematic manner.

6.2 Directions for Future Research

6.2.1 Managing Extended Transactions

From the questions posed in the introduction that prompted the development of ACTA, we were successful in addressing the first two as summarized in the previous section. In the future, we would like to take a closer look at the third question, namely, the mechanisms needed for managing extended transactions and achieving the desired correctness properties. Recall that ACTA does not assume any underlying transaction mechanism, conflict resolution mechanism, or recovery mechanism.

One approach is to design the most general transaction management scheme that is as flexible as ACTA, one which keeps track of dependencies, maintains views and conflict sets, and ensures that objects are accessed according to their conflict properties. In this general approach, object managers will keep track of dependencies that develop due to the execution of operations by transactions and communicate these dependencies to the transaction manager which, depending on the specific model in use, may choose to ignore certain dependencies. This allows object managers to be generic while designing model-specific transaction managers. In addition, such a general scheme will need a flexible recovery mechanism to handle the abortion of transactions. While this approach has its attractions because of its flexibility, it may not be as efficient as mechanisms that are targeted at specific transaction models.

An alternative, it is to let the primitives in ACTA suggest a set of implementation mechanisms, as in a "database toolset approach" [13] such that the implementation mechanisms are chosen from this toolset depending on the primitives that have been used to specify a given transaction model. Hence, it will be useful to utilize ACTA to identify the transaction management primitives required for a particular database application.

6.2.2 Modeling Real-Time Transactions

Real-time transactions must satisfy timing constraints, such as deadlines associated with transactions, in addition to other consistency requirements [1, 52, 55, 59].

Even though the work in ACTA has implications for transaction models used in real-time databases, its focus thus far has not been on time-constrained transactions. However, real-time transactions form an important class of extended transactions which we would like to study using ACTA. This will also test further the contention that ACTA is indeed a general and flexible framework. To this extent, we expect to be able to express in ACTA timing constraints in the same way as the other consistency constraints, i.e., in terms of *allowable*, *required*, or *proscribed* relationships between significant events.

An approach is to include time as part of the condition constraining the occurrence of a significant event. For instance, the requirement of a transaction t to commit by the deadline d could be expressed as:

$$pre(\mathsf{Commit}_t) \Rightarrow (time \leq d).$$

Another method is to implicitly incorporate time into a history and constrain the occurrences of events in a time quantified history. This means that all events are timestamped by their time of occurrence according to a global clock. In this method, the above example of the transaction t associated with deadline d could be specified as:

$$(\mathsf{Commit}_t \in H) \Rightarrow (\mathsf{Commit}_t \in H^d)$$

where H^d is the projection of history H with respect to time d, i.e., has all the events that occur until time d.

Using either method, it would be possible to formulate "temporal versions" of all the dependencies in Chapter 3, e.g., temporal commit dependency, temporal abort dependency, etc, as well as to define new ones. Note that neither approach adds new primitives to the ACTA formalism. These temporal dependencies can be used to define the correctness properties of real-time transactions.

We would like to evaluate both approaches based on the simplicity and elegance of the definitions produced by each approach. As in the case of non real-time transaction models, we would also like to probe the mechanisms for managing real-time transactions.

6.2.3 Modeling Other Complex Systems

Extended transaction models ascribe properties to transactions that make them resemble anything between threads and atomic transactions ("threads" with some specific properties). Given the capability of ACTA to capture this spectrum of interactions, specifying and reasoning about the properties of a complex system that does not have transactional properties should not be that different from capturing an extended transaction model.

For example, it would be interesting to use ACTA to specify and reason about the process structuring and behavior in a distributed system which supports process migration. In such a system, *Migrate* could be a significant event, similar to the Split event in split transactions, which terminates the current thread of execution and initiates a new one on another node in the network delegating some of the objects in the current state of the thread. Delegated objects comprise the initial state of the migrated thread on the remote node — not all objects constituting the state before migration need to be carried-over.

This leads to another line of future research in which we will explore the scope of the applicability of ACTA in non-transactional systems. That is, with respect to the spectrum of interactions, we would like to investigate to a greater extent the thread end of the spectrum using ACTA.

6.2.4 ACTA and Persistent Programming Languages

Research in *persistent languages* is interested in achieving persistence which supports objects that last longer than the execution of a program [2, 5, 6, 72, 83, 95]. Persistence is essential in the development of advanced systems. This work has led to the building of object-stores or object-bases [23, 74, 91], which incorporate a number of basic database mechanisms such as concurrency control and recovery in conjunction, in most cases, with atomic transactions. That is, there has been no effective way of specifying concurrency control and transaction processing within a persistent language. Thus, we would like to investigate the possibility of incorporating the primitives of ACTA into a persistent language such as ODE [2] (an extension to C++ involving persistence) or Napier88 [72].

Specifically, we are interested in linguistic structures to express the type of specification afforded by ACTA. For instance, such linguistic structures would allow the localization of the effects of activities and support control and object sharing through dependencies. By requiring the declaration of dependencies between events within an activity and across activities as in ConTracts [98], the semantics of an application can be made available to the system which can subsequently use them to efficiently manage these activities. For example, the semantics may be expressed in the form of *rules*. Dependencies in ACTA support such rules — in a sense, dependencies have the form of a rule, precondition-postcondition, and a compiler should be able to generate them so that they can be used by a generic transaction manager in order to exhibit model-specific behavior.

6.3 Conclusion

The specific contributions of this dissertation are:

- The development of ACTA, a formal framework for specifying extended transactions.
  - It is not *yet* another transaction model, but a model for transaction models.
  - It allows for an intuitive, yet precise, definition of extended transactions by characterizing the semantics of interactions between extended transactions (1) in terms of different types of dependencies between transactions, and (2) in terms of transactions' effects on objects (their state and concurrency status, i.e., synchronization state).
  - It expresses the transaction properties of:
    * Visibility in terms of the state of the objects visible to a transaction and the operations with respect to which conflicts need be considered.
    * Consistency in terms of *allowable*, *required*, or *proscribed* relationships between significant events.
    * Recovery in terms of the effects of significant events and *delegation*; the latter redefines the boundaries of failure relative to a set of objects.

    * Permanence by separating operation commitment and transaction commitment.

  - It allows dependencies to include significant events beyond the ubiquitous commit and abort events and thus, has the power to express interactions beyond those in atomic transactions. It is flexible enough to allow the definition of new dependencies as new significant events are associated with extended transactions.
  - It makes no assumptions about the structure of extended transactions and thus, is not restricted to hierarchically structured transactions.
  - It captures transaction inter-dependencies and the conflict relationships defined between operations on objects in a uniform manner; this simplifies reasoning about the interactions of transactions over shared objects and facilitates the use of transaction-specific semantics in determining conflicts between operations.

- The formalism
  - allows for the analysis of the correctness properties of extended transactions.
  - unifies existing transaction models making it possible to ascertain the relations between them, e.g., the similarities and differences between two transaction models.
  - facilitates the determination of whether two or more transaction models can be used in conjunction.
  - supports the development and analysis of new extended transaction models in a systematic manner. New transaction definitions can be derived by combining and/or extending the specifications of existing transaction models.
  - finally, even though we do not spell out the details in this thesis, the formalism can be used to show the correctness of a particular implementation of a transaction model by first formalizing the properties of the specific

mechanisms used in the implementation and then showing that they will maintain the correctness properties of the model.

It is our view that the formal ACTA framework forms the basis for a *theory* of extended transactions.

In the words of Butler Lampson, *"transaction processing is distributed computing that works."* [20]. New advanced applications of today and tomorrow are typically distributed. The ability of transactions to mask the effects of concurrency and failures makes them a powerful abstraction for structuring them in an understandable and reliable manner. By enhancing the understanding of transaction processing, ACTA contributes to making computing of today and tomorrow work!

BIBLIOGRAPHY

[1] Abbott, R. and Garcia-Molina, H. Scheduling Real-Time Transactions: A Performance Evaluation. In *Proceedings of the Fourteenth International Conference on Very Large Databases*, August 1988.

[2] Agrawal, R. and Gehani, N. H. ODE (Object Database and Environment): The Language and the Data Model. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, May 1990.

[3] Allchin, J. E. *An Architecture for Reliable Decentralized Systems.* PhD thesis, School of Information and Computer Science, Georgia Institute of Technology, September 1983.

[4] Allchin, J. E. and McKendry, M. S. Synchronization and Recovery of Actions. In *Proceedings of the 2nd Annual ACM Symposium on Principles of Distributed Computing*, pages 31–44, August 1983.

[5] Atkinson, M. P., Bailey, P. J., Chisholm, K. J., Cockshott, P. W., and Morrison, R. An approach to persistent programming. *The Computer Journal*, 26(4):360–365, November 1983.

[6] Atkinson, M., Chisolm, K., and Cockshott, P. PS-Algol: an Algol with a persistent heap. *ACM SIGPLAN Notices*, 17(7):24–31, July 1982.

[7] Avrunin, G., Buy, U., Corbett, J., Dillon, L., and Wileden, J. Automated analysis of concurrent systems with the constrained expression toolset. *(To appear in) IEEE Transactions of Software Engineering*, 1991.

[8] Badrinath, B. *Concurrency control in complex information systems: A semantics-based approach.* PhD thesis, Univeristy of Massachusetts, Amherst, MA, August 1989.

[9] 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, Atlantic City, NJ, May 1990.

[10] Badrinath, B. and Ramamritham, K. Semantics-based concurrency control: Beyond Commutativity. *(to appear in) ACM Transactions on Database Systems*, 1991.

[11] Bancilhon, F., Kim, W., and Korth, H. A model of CAD Transactions. In *Proceedings of the Eleventh International Conference on Very Large Databases*, pages 25–33, Stockholm, August 1985.

[12] Barron, J. Dialogue and Process Design for Interactive Information Systems Using TAXIS. In *Proceedings of the Conference on Office Information Systems*, pages 12–20, June 1982.

[13] Batory, D. S. GENESIS: A project to develop an Extensible Database Management System. In *Proceedings of the International Workshop on Object-Oriented Database Systems*, pages 207–208. IEEE Computer Society, September 1986.

[14] Batory, D. S. and Kim, W. Modeling concepts for VLSI CAD objects. *ACM Transactions on Database Systems*, 10(3):322–346, September 1985.

[15] Beeri, C., Bernstein, P. A., and Goodman, N. A Model for Concurrency in Nested Transaction Systems. *Journal of the ACM*, 36(2):230–269, April 1989.

[16] Bernstein, P. A. and Goodman, N. Concurrency control in distributed database systems. *ACM Computing Surveys*, 13(2):185–221, June 1981.

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

[18] Birman K. P. et al. Implementing fault-tolerant distributed objects. *IEEE Transactions on Software Engineering*, 11(6):520–530, June 1985.

[19] Bjork, L. and Davis, C. The Semantics of the Preservation and Recovery of Intergrity in a Data System. Technical report, IBM Technical Report TR 02.540, December 1972.

[20] Black, A. Understanding Transactions in the Operating System Context. *ACM Operating System Review*, 25(1):73–76, January 1991.

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

[22] Campbell, R. and Habermann, A. *The specification of process synchronization by Path Expressions*, volume 16 of *Lecture Notes in Computer science.* Spring-Verlag, 1974.

[23] Carey, M. J., DeWitt, D. J., Richardson, J. E., and Shekita, E. J. Object and file management in the EXODUS extensible database system. In *Proceedings of the Twelfth International Conference on Very Large Data Bases*, pages 91–100, Kyoto, Japan, August 1986. Morgan Kaufmann.

[24] Carey, M. J., DeWitt, D. J., Richardson, J. E., and Shekita, E. J. Object and file management in the EXODUS extensible database system. In *Proceedings of the Twelfth International Conference on Very Large Databases, Kyoto, Japan*, August 1986.

[25] Carey M. et al. The architecture of the EXODUS extensible DBMS. In Stonebraker, M., editor, *Readings in Database Systems*, pages 488–501. Morgan Kaufmann, 1988.

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

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

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

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

[30] Copeland, G. and Maier, D. Making smalltalk a database system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 316–325, Boston, MA, June 1984.

[31] Davies, C. T. Data Processing Spheres of Control. *IBM System Journal*, 17(2), February 1978.

[32] Davis, C. Recovery Semantics for a DB/DC System. In *Proceedings of the 28th ACM National Conference*, pages 136–141, August 1973.

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

[34] Dayal, U. and Smith, J. PROBE: A knowledge-oriented database management system. In *Proceedings of the Islamorada Workshop on Large Scale Knowledge Base and Reasoning Systems*, February 1985.

[35] Dillon, L., Avrunin, G., and Wileden, J. Constrained expressions: Toward broad applicability of analysis methods for distributed software systems. *ACM Transaction on Programming Languages and Systems*, 10(3):374–402, July 1988.

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

[37] Elmagarmid, A., Leu, Y., Litwin, W., and Rusinkiewicz, M. A Multidatabase Transaction Model for InterBase. In *Proceedings of the Sixteenth International Conference on Very Large Databases*, pages 507–518, August 1990.

[38] Elmagarmid A. (Editor). *Database Transaction Models for Advanced Applications.* Morgan Kaufmann, 1991.

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

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

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

[42] Garcia-Molina, H. Using semantic knowledge for transaction processing in a distributed database. *ACM Transactions on Database Systems*, 8(2), June 1983.

[43] Garcia-Molina, H., Gawlick, D., Klein, J., Kleissner, K., and Salem, K. Coordinating Multi-Transaction Activities. Technical Report CS-TR-247-90, Dept. of Computer Science, Princeton University, February 1990.

[44] 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, March 1991.

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

[46] Garza, J. F. and Kim, W. Transaction management in an Object-Oriented database system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 37–45, Chicago, Illinois, June 1988.

[47] Gray, J. The Transaction Concept: Virtues and Limitations. In *Proceedings of the Seventh International Conference on Very Large Databases*, pages 144–154, September 1981.

[48] Gray, J. N., Lorie, R. A., Putzulo, G. R., and Traiger, I. L. Granularity of locks and degrees of consistency in a shared database. In *Proceedings of the First International Conference on Very Large Databases*, pages 25–33, Framingham, MA, September 1975.

[49] Gray, J. N., Lorie, R. A., Putzulo, G. R., and Traiger, I. L. Granularity of locks and degrees of consistency in a shared database. In Stonebraker, M., editor, *readings in Database Systems*, pages 94–121. Morgan Kaufmann, 1988.

[50] Hadzilacos, T. and Hadzilacos, V. Transaction synchronization in object bases. In *Proceedings of the Seventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, Austin, Texas*, March 1988.

[51] Haerder, T. and Reuter, A. Principles of transaction-oriented database recovery. *ACM Computing Surveys*, 15(4):287–317, December 1983.

[52] Haritsa, J., Carey, M., and Livny, M. On Being Optimistic about Real-Time Constraints. In *Proceedings of the Nineth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 1990.

[53] Herlihy, M. P. Extending multiversion timestamping protocols to exploit type information. *IEEE Transactions on Computers*, 35(4):443–449, April 1987.

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

[55] Huang, J., Stankovic, J., Ramamritham, K., and Towsley, D. Performance Evaluation of Real-Time Optimistic Concurrency Control Schemes. In *Proceedings of the seventeenth International Conference on Very Large Databases*, September 1991.

[56] Kim, W., Banerjee, J., Chou, H.-T., Garza, J., and Woelk, D. Composite object support in an Object-Oriented database system. In *Proceedings of Object-Oriented Programming Systems, Languages, and Applications*, pages 118–125, Orlando, Florida, October 1987.

[57] Korth, H. F., Kim, W., and Bancilhon, F. On Long-Duration CAD Transactions. *Information Sciences*, 46(1-2):73–107, October-November 1988.

[58] Korth, H. F., Levy, E., and Silberschatz, A. Compensating Transactions: A New Recovery Paradigm. In *Proceedings of the Sixteenth International Conference on Very Large Databases*, pages 95–106, Brisbane, Australia, August 1990.

[59] Korth, H. F., Soparkar, N., and Silberschatz, A. Triggered Real-Time Databases with Consistency Constraints. In *Proceedings of the Sixteenth International Conference on Very Large Databases*, Brisbane, Australia, August 1990.

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

[61] Korth, H. F. Locking primitives in a database system. *Journal of the ACM*, 30(1):55–79, January 1983.

[62] Levy, E., Korth, H. F., and Silberschatz, A. A Theory of Relaxed Atomicity. In *Proceedings of the 10th Annual ACM Symposium on Principles of Distributed Computing*, Montreal, Canada, August 1991.

[63] Liskov, B. and Scheifler, R. Guardians and Actions: Lingusitic support for robust distributed programs. *ACM Transactions on Programming Languages and Systems*, 8(4):484–502, December 1983.

[64] Lynch, N. A. Multilevel atomicity — A new correctness for database concurrency control. *ACM Transactions on Database Systems*, 8(4):484–502, December 1983.

[65] Maier, D., Stein, J., Ottis, A., and Purdy, A. Development of an Object-Oriented DBMS. In *Proceedings of Object-Oriented Programming Systems, Languages, and Applications*, pages 472–482, Portland, Oregon, October 1986.

[66] Manola, F. and Dayal, U. PDM: An object-oriented data model. In *Proceedings of the International Workshop on Object-Oriented Database Systems*, pages 18–25. IEEE Computer Society, September 1986.

[67] Martin, B. E. Scheduling protocols for nested objects. Technical Report CS-094, Department of Computer Science and Engineering, University of California, San Diego, California, 1988.

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

[69] Milner, R. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer science.* Spring-Verlag, 1980.

[70] Milner, R. *Communication and Concurrency.* Prentice-Hall, 1989.

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

[72] Morrison, R., Brown, A., Carrick, R., Connor, R., Dearle, A., and Atkinson, M. P. The Napier Type System. In *Proceedings of the Workshop on Persistent Object Systems: Their Design, Implementation, and Use*, pages 253–269, 1989.

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

[74] Moss, J. E. B. Design of the Mneme persistent object store. *ACM Transactions on Office Information Systems*, 8(2):103–139, April 1990.

[75] Moss, J. E. B., Griffeth, N., and Graham, M. Abstraction in recovery management. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 72–83, May 1986.

[76] Murata, T. Petri Nets: Properties, Analysis and Applications. *Proceedings of IEEE*, 77(4):541–580, April 1989.

[77] Papadimitriou, C. H. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, October 1979.

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

[79] Pratt, V. Modeling concurrency with partial orders. *International Journal on Parallel Programming*, 15(1):33–71, 1986.

[80] Pu, C. *Replication and Nested Transanctions in the Eden Distributed System.* PhD thesis, University of Washington, 1986.

[81] Pu, C. From Nested Transactions to Supertransactions. *Bulletin of the IEEE Technical Committee on Data Engineering*, 10(3):19–25, September 1987.

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

[83] Richardson, J. E. and Carey, M. J. Persistence in the e language: Issues and implementation. *Software: Practice and Experience*, 19(12):1115–1150, December 1990.

[84] Schwarz, P. M. *Transactions on Typed Objects.* PhD thesis, Carnegie-Mellon University, Pittsburgh, PA, December 1984.

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

[86] Schwarz P. M. et al. Extensibility in the starburst system. In *Proceedings of the International Workshop on Object-Oriented Database Systems*, pages 85–92. IEEE Computer Society, September 1986.

[87] Sha, L. *Modular concurrency control and failure recovery— Consistency, Correctness and Optimality.* PhD thesis, Department of Computer and Electrical Engineering, Carnegie-Mellon University, 1985.

[88] Shrivastava, S. K. and Wheater, S. M. Implementing fault-tolerant distributed applications using objects and multi-coloured actions. In *Proceedings of the 10th International Conference on Distributed Computing Systems*, Paris, France, May 1990.

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

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

[91] Skarra, A., Zdonik, S. B., and Reiss, S. P. An object server for an object oriented database system. In Dittrich, K. and Dayal, U., editors, *Proceedings of the International Workshop on Object-Oriented Database Systems*, pages 196–204, Pacific Grove, Calif., September 1986. IEEE Computer Society Press, Washington, DC.

[92] Spector A. Z. et al. Support for distributed transactions in the TABS prototype. *IEEE Transactions on Software Engineering*, 11(6):520–530, June 1985.

[93] Spector A. Z. et al. Camelot: A Distributed Transaction Facility for Mach and the Internet - An Interim Report. Technical Report CMU-CS-87-129, Carnegie-Mellon University, June 1987.

[94] Stonebraker, M. and Rowe, L. The design of POSTGRES. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 340–355, May 1986.

[95] Straw, A., Mellender, F., and Riegel, S. Object management in a persistent Smalltalk system. *Software: Practice and Experience*, 19(8):719–737, August 1989.

[96] Verhofstad, J. Recovery Techniques for Database Systems. *ACM Computing Surveys*, 10(2):167–196, June 1978.

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

[98] Wächter, H. and Reuter, A. The ConTract Model. In Elmagarmid, A. K., editor, *Database Transaction Models for Advanced Applications*. Morgan Kaufmann, 1991.

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

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

[101] Yodaiken, V. The algebraic feedback product of automata. In *Papers from the DIMACS Workshop on Computer Aided Verification*, AMS-DIMACS Series. American Mathematical Society, 1991.