A Taxonomy of Correctness Criteria in Database Applications [1] [2]

Krithi Ramamritham

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

(413) 545-0196
krithi@cs.umass.edu

Panos K. Chrysanthis

Dept. of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260

(412) 624-8924
panos@cs.pitt.edu

Abstract

Whereas serializability captures *database consistency requirements* and *transaction correctness properties* via a single notion, recent research has attempted to come up with correctness criteria that view these two types of requirements independently. The search for more flexible correctness criteria is partly motivated by the introduction of new transaction models that extend the traditional atomic transaction model. These extensions came about because the atomic transaction model in conjunction with serializability is found to be very constraining when used in advanced applications, such as design databases, that function in distributed, cooperative, and heterogeneous environments.

In this paper, we develop a taxonomy of various *correctness criteria* that focus on database consistency requirements and transaction correctness properties from the viewpoint of *what* the different dimensions of these two are. This taxonomy allows us to categorize correctness criteria that have been proposed in the literature. To help in this categorization, we have applied a uniform specification technique, based on ACTA, to express the various criteria. Such a categorization helps shed light on the similarities and differences between different criteria and to place them in perspective.

Keywords: Transaction processing, concurrency control, database correctness criteria, formal specifications.

---

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

[2]A preliminary version of this paper was presented at the International Workshop on Distributed Object Management, Edmonton, Canada, in August 1992.

Contents

1 Introduction

Database consistency requirements capture correctness from the perspective of objects in the database – as transactions perform operations on the objects. On the other hand, *transaction correctness properties* capture correctness from the perspective of the structure and behavior of transactions. For example, they deal with the results of transactions and the interactions between transactions. Serializability [Eswaran, 1976] captures database consistency requirements and transaction correctness properties via a single notion: (1) The state of the database at the end of a set of concurrent transactions is the same as the one resulting from some serial execution of the same set of transactions; (2) The results of transactions and the interactions among the set of transactions are the same as the results and interactions, had the transactions executed one after another in this serial order. As applications using databases become more complex, the *correctness criteria* that are *acceptable* to the application become more complex and hence harder to capture using a single correctness notion.

Recent research has attempted to come up with correctness criteria that view these two types of requirements independently. An early example is nested transactions [Moss, 1981], in which the database consistency requirements are captured by requiring the serializability of independent (sub)transactions; additional transaction structural properties specify the correctness of subtransactions of individual nested transactions. The search for more flexible correctness requirements is further motivated by the introduction of other transaction models that extend the traditional atomic transaction model. (See [Elmagarmid, 1992] for a description of some of the *extended transaction models*.) These extensions came about because the atomic transaction model in conjunction with serializability is found to be very constraining when applied in advanced applications such as design databases that function in distributed, cooperative, and heterogeneous environments [Barghouti, 1991, Korth, 1988].

Proposed correctness criteria range from the standard serializability notion to eventual consistency [Sheth, 1990]. Quasi-serializability [Du, 1989], predicatewise serializability [Korth, 1988b], etc., are points that lie within this range. *Eventual consistency* can be viewed as a "catch-all" term with different connotations: For example, requiring consistency "at a specific real-time", "within some time" or "after a certain amount of change to some data", or enforcing consistency "after a certain value of the data is reached", etc. Whereas serializability and its relaxations are, in general, application and transaction model independent criteria, eventual consistency, as the examples above show, are application and transaction model specific. It is not difficult to see that these relaxed correctness requirements are useful within a single database as well as in multi-database environments.

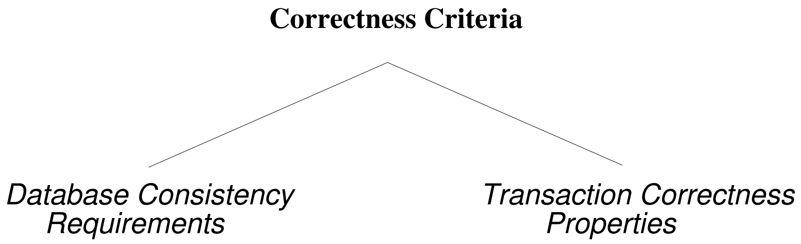
Whereas serializability works under the simple assumption that individual transactions maintain the consistency of the database, proposed correctness criteria require more from the transaction developers. In particular, a transaction may have to be aware of the functionality of other transactions and the potential interactions among transactions, especially in a cooperative environment. This makes transaction development as well as management more difficult. Our goal in this paper is to understand the conceptual similarities and differences between different correctness criteria without getting into the practical implications of adopting them.

So, in this paper we examine database consistency constraints and transaction correctness properties from the viewpoint of *what* the different dimensions of these two types of correctness are. This taxonomy allows us to categorize existing proposals thereby shedding some light on the similarities and differences between the proposals and to place them in perspective. The categorization also helps us determine whether or not a correctness notion is transaction model specific or application specific. We will see that even though some of the correctness notions were motivated by specific transaction models or specific applications, they have broader applicability.

To help in this categorization, we apply a uniform specification technique to express the various correctness criteria that have been proposed. The technique is based on the ACTA formalism [Chrysanthis, 1990, 1991] which heretofore has been used for the specification of and for reasoning about extended transactions. One of the key ingredients of ACTA is the idea of constraining the occurrence of *significant events* associated with transactions, *Begin*, *Abort*, and *Split*, for example. These constraints are expressed in terms of necessary and sufficient conditions for events to occur. These, in turn, relate to the ordering of events and the validity of relevant conditions. Such constraints can also facilitate the specification of database consistency requirements and transaction correctness properties. The ACTA formalism is introduced in Section 3.

The rest of the paper is structured as follows: Subsection 2.1 provides a taxonomy of database consistency requirements while 2.2 provides a taxonomy of transaction correctness properties. A specification of existing proposals as well as their categorization (based on the taxonomy) is the subject of Section 4. Section 5 concludes the paper with some discussions of the next step in this work.

2 A Taxonomy of Correctness Criteria



In this section we study the different dimensions of the two aspects of correctness, namely, consistency of database state and correctness of transactions, in order to develop a taxonomy of correctness criteria. For concreteness, we give examples as the taxonomy is developed.

2.1 Database Consistency Requirements

Database consistency requirements can be examined with respect to two issues with further divisions of each as discussed below (see Figure 1).

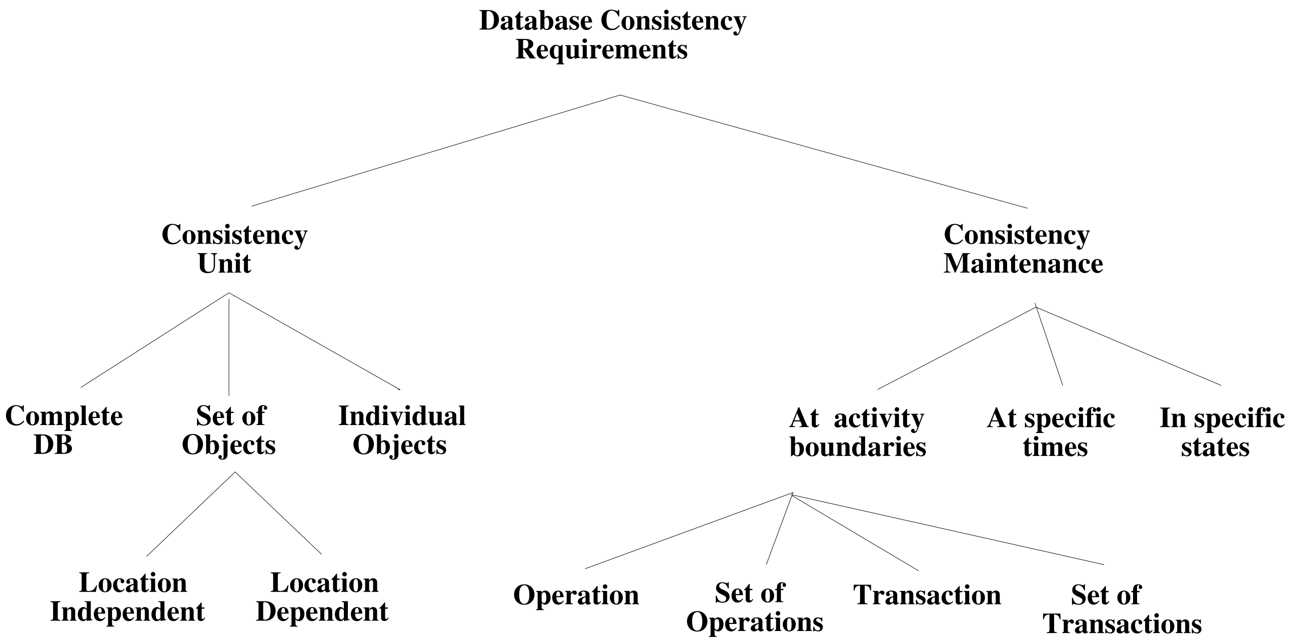


Figure 1: Dimensions of Database Consistency

Consistency Unit:

This is related to the data items involved in a consistency requirement.

- *Complete Database*:

  All the objects in the database have to be consistent locally as well as mutually consistent, i.e., they should satisfy all the database integrity constraints typically specified in the form of predicates on the state of the objects. Semantics of the objects can be taken into account to improve concurrent access to the objects while maintaining consistency [Chrysanthis, 1991a].

  Example: Traditional serializability (SR) applied to atomic transactions [Bernstein, 1987].

- *Subsets of the objects in the database*:

  - *Location-independent subsets*:

    The database is viewed as being made up of subsets of objects. The subsets are not necessarily disjoint and may be statically or dynamically defined. Each object in the database is expected to be consistent locally but mutual consistency is required only for objects that are within the same subset.

    Example: Setwise serializability (SSR) applied to *compound transactions* [Sha, 1985] and Predicatewise serializability (PSR) applied to cooperative transactions [Korth, 1988].

  - *Location-dependent subsets*:

    Each subset corresponds to one of the sites of a (distributed / heterogeneous) database. In addition to mutual consistency among objects in a subset (i.e., site), consistency among subsets is also required depending on which parts of a database are accessed by a transaction.

    Example: Quasi-serializability (QSR) [Du, 1989] and its generalization [Mehrotra, 1991] applied to distributed transactions.

- *Individual Objects*:

  Each object in the database is expected to be consistent locally.

  Example: Linearizability [Herlihy, 1987] applied to objects accessed by concurrent *processes*.

Consistency Maintenance:

This is related to the issues of *when* a consistency requirement is expected to hold.

- *At activity boundaries* (an activity denotes a unit of work):
  The activity is allowed to complete only if the requirement holds, i.e., completion is delayed until consistency holds. If an activity cannot complete successfully while maintaining consistency, it is aborted or compensated.

  Example: SR, PSR, QSR, and cooperative serializability (CoSR), applied to atomic, nested, and distributed transactions.

  Depending on what the activity is, we can further develop the taxonomy.

  - *When an operation completes*:
    When an operation on an object completes, the necessary consistency specifications must hold.

    Example: Concurrent processes accessing shared objects.
  - *When a set of operations completes*:
    When a set of operations performed by a transaction completes, the necessary consistency is expected to hold.

    Example: Semantic atomicity [Garcia-Molina, 1983] and multilevel atomicity [Lynch, 1983].
  - *When a transaction completes*:
    Consistency is expected to hold upon a transaction's completion.

    Example: Atomic transactions.
  - *When a set of transactions completes*:
    Consistency is expected to hold not when individual transactions complete but when a set of transactions completes.

    Example: Cooperative transactions [Korth, 1988b], sagas [Garcia-Molina, 1987].
- *At specific points of time*:
  Consistency between related objects is maintained in a deferred manner only at/after specific points in time. This is an example of *temporal consistency* [Sheth, 1990].

  Example: A bank account is expected to be made consistent, with respect to the debits and credits that occur on a given day, upon closing of business.

- *In specific states*:
  Objects may be required to be mutually consistent only when a certain number of updates have been made to one of the objects or a state satisfying a certain predicate is reached. Thus, in this case also, consistency between related objects is maintained in a deferred manner.

  Example: A centralized database of a department store chain may require updates only upon the completion of 100 sales at a particular store. Such requirements are referred to as *spatial consistency* in [Sheth, 1990].

2.2 Transaction Correctness Properties

As was mentioned in the introduction, serializability suffices as a correctness criterion for traditional atomic transactions since once *individual transactions are guaranteed to take one consistent database state to another consistent state*, serializability guarantees that a set of concurrent transactions when started in a consistent state take the database to another consistent state. So the only transaction correctness property of interest is: Each transaction when executed by itself must maintain database consistency. From this it follows that, under serializability, the output of a transaction reflects a consistent database state. However, more elaborate correctness properties have been proposed in the context of additional application requirements and newer transaction models. These transaction correctness properties can be discussed with respect to four criteria (see Figure 2):

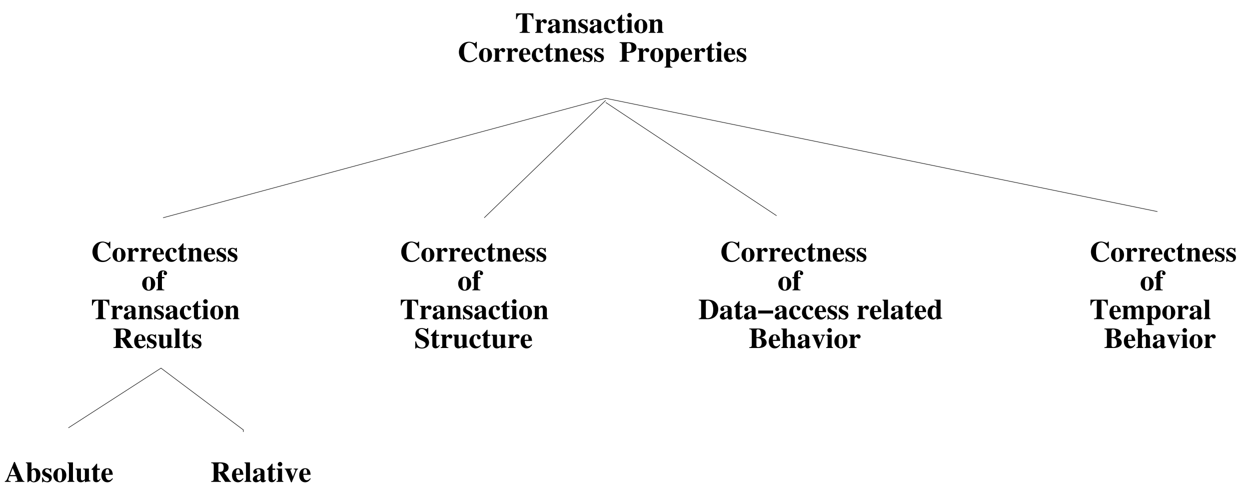


Figure 2: Dimensions of Transaction Correctness

- *Correctness of transaction results*:
  - *Absolute*:
    The output of transactions must reflect a consistent database state.

    Example: SR applied to atomic transactions, QSR applied to distributed transactions.
  - *Relative*:
    Outputs of a transaction are considered correct even if they do not reflect a consistent state of the object, as long as they are within a certain bound of the result that corresponds to the consistent state.

    Example: Epsilon-serializability (ESR) [Pu, 1991] applied to Epsilon-transactions, approximate query processing [Hou, 1989].
- *Correctness of transaction structure*:
  Correctness depends on the (structural) relationship between transactions. This is typically specified in terms of prescribed and/or proscribed *commit, abort, begin*, and other types of dependencies [Chrysanthis, 1991] between transactions. Since structural properties are governed by a particular transaction model, the specifications of the model express these requirements.

  Example: Sagas [Garcia-Molina, 1987], multi-level serializability [Korth, 1990c].
- *Correctness of data access related transaction behavior*:
  Transactions are required to perform operations on objects in a certain manner to be considered correct. That is, these requirements are constraints on the history of concurrent operations.

  Example: To satisfy serializability, conflict relationship between transactions – as they access data concurrently – must be acyclic. Patterns [Skarra, 1991] are more application specific correctness requirements that reflect the (semantics of) usage of the object.
- *Correctness of temporal behavior of transactions*:
  Transactions have start time and completion time (deadline) constraints.

  Example: Transactions in real-time systems.

The taxonomy just presented shows how the various correctness requirements can be viewed from the perspectives of database consistency and transaction correctness. It is perhaps clear that, from a database application designer's perspective, what is required is to specify which leaves of the taxonomy correspond to his/her application and then provide additional specifications required by the individual leaves. For instance, if correctness depends on transactions' structural properties, additional specifications will be needed to specify what these properties are. For example, if transactions in an application behave according to the nested transaction model, an axiomatic specification of the nested transaction model [Chrysanthis, 1991] will supplement the identification of the fact that transactions have structure related correctness requirements.

We revisit the correctness notions in Section 4.1 where serializability related correctness notions are formally specified and categorized along the different dimensions of the taxonomy. Sections 4.2 and 4.3 deal with the formal specification of more general correctness criteria that are not directly related to serializability but deal, for example, with transaction structure and behavior, specific states of objects, or specific times.

3 A Quick Introduction to The Acta Formalism

ACTA is a first-order logic based formalism. As mentioned earlier, the idea of significant events underlies ACTA's specifications. Section 3.1 discusses these events. Specifications involve constraints on the occurrence of individual significant events as well as on the history of occurrence of these events. Hence the notion of history and the necessary and sufficient conditions for the occurrence of significant events are introduced in Section 3.2. Finally, Section 3.3 shows how sharing of objects leads to transaction inter-relationships which in turn induces certain dependencies between concurrent transactions.

3.1 Significant Events Associated with Transactions

During the course of their execution, transactions invoke operations on objects. Also, they 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 [Chrysanthis, 1991]. 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. Other transaction management primitives include *Spawn*, found in the nested transaction model [Moss, 1981], *Split*, found in the split transaction model [Pu, 1988], and *Join*, a transaction termination event also found in the split transaction model.

DEFINITION 3.1: Invocation of a transaction management primitive is termed a *significant event*. A transaction model defines the significant events that transactions adhering to that model can invoke.

The set of events invoked by a transaction t is a partial order with ordering relation $\rightarrow$ where $\rightarrow$ denotes the temporal order in which the related events occur.

$ts(\epsilon)$ gives the time of occurrence of event ϵ according to a globally synchronized clock[3]. Clearly, $ts(\beta)$ will be larger than $ts(\alpha)$ if α appears earlier in the partial order ($\alpha \rightarrow \beta$). Further, no two significant events *that relate to the same transaction* can occur with the same ts value.

3.2 History, Projection of the History, and Constraints on Event Occurrences

The concurrent execution of a set of transactions T is represented by the *history* [Bernstein, 1987] of the events invoked by the transactions in the set T and indicates the (partial) order in which these events occur. The partial order of the operations in a history is consistent with the partial order of the events of each individual transaction t in T.

The *projection* of a history H is a subhistory that satisfies a given criterion. For instance,

- The projection of a history H with respect to a specific transaction t yields a subhistory with just the events invoked by t. This is denoted by H_t.
- The projection of a history H with respect to a specific time interval [i, j] yields the subhistory with the the events which occurred between i and j (inclusive) and is denoted by $H^{[i,j]}$.

  When i = system initiation time, we drop the first element of the pair. Thus $H^j = H^{[system\_init\_time,j]}$ denotes all the events that occur until time j.

Consistency requirements imposed on concurrent transactions executing on a database can be expressed in terms of the properties of the resulting histories.

[3]This is obviously as abstraction – the effects of realizing this by a set of closely synchronized clocks on individual nodes in a distributed system will not be discussed here.

The occurrence of an event in a history can be constrained in one of three ways: (1) An event ϵ can be constrained to occur *only after* another event ϵ'; (2) An event ϵ can occur *only if* a condition c is true; and (3) a condition c can *require* the occurrence of an event ϵ.

DEFINITION 3.2: The predicate $\epsilon \rightarrow \epsilon'$ is true if event ϵ *precedes* event ϵ' in history H. It is false, otherwise. (Thus, $\epsilon \rightarrow \epsilon'$ implies that $\epsilon \in H$ and $\epsilon' \in H$.)

DEFINITION 3.3: $\epsilon \in H \Rightarrow Condition_H$, where $\Rightarrow$ denotes *implication*, specifies that the event ϵ can belong to history H *only if* $Condition_H$ is satisfied. In other words, $Condition_H$ is *necessary* for ϵ to be in H. $Condition_H$ is a predicate involving the events in H.

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

DEFINITION 3.4: $Condition_H \Rightarrow \epsilon \in H$ specifies that if $Condition_H$ holds, ϵ should be in the history H. In other words, $Condition_H$ is *sufficient* for ϵ to be in H.

We now describe some common types of constraints.

1. $Commit_{t_j} \in H \Rightarrow (Commit_{t_i} \in H \Rightarrow (Commit_{t_i} \rightarrow Commit_{t_j}))$. This says that if both transactions t_i and t_j commit then the commitment of t_i precedes the commitment of t_j. This **Commit-Dependency** is indicated by $(t_j\ \mathcal{CD}\ t_i)$. In general, $(Commit_{t_j} \in H \Rightarrow condition)$ specifies that *condition* should hold for t_j to commit.

2. $Abort_{t_i} \in H \Rightarrow Abort_{t_j} \in H$ i.e., if t_i aborts then t_j aborts, states the **Abort-Dependency** of t_j on t_i $(t_j\ \mathcal{AD}\ t_i)$. In general, $(condition \Rightarrow Abort_{t_j} \in H)$ specifies that if *condition* holds t_j aborts.

3. $Begin_{t_j} \in H \Rightarrow (Begin_{t_i} \rightarrow Begin_{t_j})$ states that transaction t_j cannot begin executing until transaction t_i has begun.

3.3 Objects, Operations, and Conflicts

A transaction accesses and manipulates the objects in the database by invoking operations specific to individual objects. It is assumed that an operation always produces an output (return value), that is, it has an outcome (condition code) or a result. The result of an operation on an object depends on the current state of the object. For a given state s of an object, we use $return(s, p)$ to denote the output produced by operation p, and $state(s, p)$ to denote the state produced after the execution of p.

DEFINITION 3.5: Invocation of an operation on an object is termed an *object event.* The type of an object defines the object events that pertain to it. We use $p_t[ob]$ to denote the object event corresponding to the invocation of the operation p on object ob by transaction t. Object events are also part of the history H.

DEFINITION 3.6: Let $H^{(ob)}$ denote the projection of the H with respect to the operations on ob. 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.*

($\circ$ denotes the composition of operations; $H \circ p$ appends p to history H.) Thus, two operations conflict if their effects on the state of an object are not independent of their execution order (first clause) or their return values are not independent of their execution order (second and third clauses). From now on, we drop the first parameter of conflict, namely, $H^{(ob)}$.

4 Specification and Categorization of Correctness Criteria

In this section, we study various database consistency requirements and transaction correctness properties that have been proposed and place them in perspective, given the taxonomy of the previous section. Broadly speaking, Section 4.1 deals with transaction model and application independent correctness criteria (even though, as we will see, those who proposed them may have had a specific transaction model or application in mind), Section 4.2 discusses transaction model dependent but application independent criteria, and Section 4.3 examines transaction and application dependent consistency requirements. (For a complete axiomatic semantics of the various extended transaction models, the reader is referred to [Chrysanthis, 1991].)

4.1 Transaction and Application Independent Criteria

In general, transaction and application independent correctness criteria are extensions to serializability. In this section, we first specify some of these extensions using the formalism described in the previous section and then use the specifications to show how the different extensions relate to each other. All of these criteria are based on the notion of *conflicts* and their preservation in equivalent histories. Thus, we do not discuss correctness criteria such as *view serializability* [Yannakakis, 1984] that are not as easy to realize.

DEFINITION 4.7: Let $\mathcal{R}$ be a binary relation on a set of transactions T, t_i, $t_k \in T$, $t_i \neq t_k$. $\mathcal{R}^*$ is the transitive-closure of $\mathcal{R}$; i.e.,

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

4.1.1 Serializability

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

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

Let H be the history of events relating to committed transactions in T. That is, H is the projection of the system's history with respect to committed transactions in T.

DEFINITION 4.8: $\forall\ t_i, t_j \in T, t_i \neq t_j$,

$$(t_i\ \mathcal{C}\ t_j) \text{ if } \exists ob\ \exists p, q\ (\mathit{conflict}(p_{t_i}[ob], q_{t_j}[ob]) \wedge (p_{t_i}[ob] \rightarrow q_{t_j}[ob])).$$

DEFINITION 4.9:

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

To illustrate the practical implications of these definitions, note that the conflict relation $\mathcal{C}$ captures the fact that two transactions have invoked conflicting operations on the same object and the order in which they have invoked the conflicting operations. Consequently, the $\mathcal{C}$ relation captures direct conflicts between transactions in a history as well as their serialization order. The fact that a serialization order is acyclic is stated by requiring that there be no cycles in the $\mathcal{C}$ relation.

Note also that the above definitions do not involve any significant events. This reflects the fact that serializability per se does not constrain the occurrence of any significant event, e.g., a *Commit* event to happen only after another *Commit* event. (If the $\mathcal{C}$ relationship between transactions is acyclic, transactions in H can commit in any order.) That is, the *commit order* of transactions is not necessarily the same as their serialization order and hence, the commit order cannot be used to induce the serialization order. However, a commit order induced by a $\mathcal{C}$ relation is consistent with the serialization order. For example, consider the case of *rigorous histories* [Breitbart, 1991] such as the ones produced by the strict two-phase locking protocol [Eswaran, 1976]. In this case, if transactions t_i and t_j have a $\mathcal{C}$ relationship, i.e., they have invoked conflicting operations, a commit dependency [Chrysanthis, 1991] forms between t_i and t_j. (Conflicting operations may also produce abort dependencies between the invoking transactions; but an abort dependency implies a commit dependency.) By requiring that the $\mathcal{C}$ relation be acyclic, commit dependencies must also be acyclic. By inducing a commit dependency between every pair of transactions invoking conflict operations, the commit order specified by the commit dependencies is the same as the serialization order.

With respect to the taxonomy of Section 2, for serializability, the consistency unit is the complete database, and consistency is required at transaction boundaries. Absolute correctness of transaction results is expected. Atomic transactions and top-level transactions of nested transactions, for example, behave according to the serializability correctness criterion.

The semantics of the operations on the objects (for example, see [O'Neil, 1986, Herlihy, 1987, Badrinath, 1990]) can be used to define the *conflict* relationship between operations. Furthermore, different *degrees* of consistency [Gray, 1975] can be ensured by ignoring some of the conflicts. The resulting inconsistencies can be accommodated in applications that can cope with such inconsistencies or when these are masked by the structuring of the objects used by the applications. The former is the case in [Gray, 1975] and with ESR [Pu, 1991] (see [Ramamritham, 1993b] for a formal characterization of ESR). The latter is the case with abstract serializability – used in the context of multi-level transactions [Weikum, 1984, Moss, 1986, Martin, 1988, Beeri, 1989, Badrinath, 1990].

4.1.2 Predicatewise Serializability

Predicatewise serializability has been proposed in [Korth, 1988, 1988b] 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.

Let $P = (P_1 \wedge P_2 \ldots \wedge P_n)$ be the database consistency constraint. Suppose the disjunctive clause P_k relates to objects in $D_k \subseteq DB$, where DB is the database.

$\forall k \in \{1 \ \ldots \ n\}$, let $\mathcal{C}_k$ be a binary relation on transactions in T.

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

Definition 4.10: $\forall k \in \{1 \ \ldots \ n\} \ \forall \ t_i, t_j \in \ T, \ t_i \neq t_j$

$$(t_i \ \mathcal{C}_k \ t_j) \quad \text{if} \quad \exists ob \in D_k \ \exists p, q \ (\mathit{conflict}(p_{t_i}[ob], q_{t_j}[ob]) \wedge (p_{t_i}[ob] \rightarrow q_{t_j}[ob])).$$

Definition 4.11:

H is *predicatewise serializable* iff $\forall t \in T\ \forall D_k,\ 1 \leq k \leq n\ \neg(t\ \mathcal{C}_k^*\ t)$.

In [Sha, 1985], each D_k is said to be an *atomic data set*. With respect to the taxonomy, for predicatewise serializability, an atomic data set [Sha, 1985] forms a consistency unit and consistency is required at transaction boundaries. Absolute correctness of transaction results is expected. Compound transactions [Sha, 1985] behave according to the predicatewise serializability correctness criterion.

4.1.3 Cooperative Serializability

We define *cooperative serializability* (CoSR) with respect to a set of transactions which maintain some consistency properties. Transactions form cooperative transaction sets. A cooperative transaction set could be formed by 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, a set of cooperative transactions becomes the unit of concurrency with respect to serializability.

Let T_c be a set of cooperative transactions, $T_c \subseteq T$.

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

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

Definition 4.12: $\forall\ t_i,\ t_j,\ t_k\ \in\ T, t_i \neq t_j, t_i \neq t_k, t_j \neq t_k\ \forall T_c \subseteq T$

$(t_i\ \mathcal{C}_c\ t_j)$, if

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

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) serializability (definition 4.8). 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 4.13:

H is *cooperative serializable* iff $\forall t \in T \; \neg(t \; \mathcal{C}_c^* \; t)$.

With respect to the taxonomy of the previous section, for cooperative serializability, the consistency unit is the complete database, consistency is required when an ordinary transaction (not a member of a T_c) completes or a set of cooperating transactions complete. Absolute correctness of transaction results is expected. The correctness requirement expressed informally in [Martin, 1992] corresponds to CoSR.

Note that if each T_c is a singleton set, then no cooperation occurs and $\mathcal{C}_c$ is equivalent to $\mathcal{C}$. In addition, cooperative serializability does not imply that all transactions in a coperative set must commit or none. For example, let us consider the notion of *Multi-database serializability (MSR)* [Mehrotra, 1991, 1992] which has been proposed to deal with correctness of transactions in multidatabase systems, e.g., 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). Specifically, MSR is defined in the context of emulating two-phase commit protocols in multidatabases using redo transactions. The idea is that the commitment of a global transaction can be decided using the 2PC protocol between the Multidatabase agents that interface the local DBMS without the participation of the local DBMS and hence, a subtransaction is not required to enter the prepare to commit state during the decision phase. If the subtransaction is aborted but the final decision is to commit the global transaction, the updates of the aborted subtransaction are performed subsequently by a redo transaction. This implies that (1) the state of the database against which the redo transaction executes, should be the same as the one seen by the aborted subtransaction and (2) the redo transaction should not invalidate any other active or committed (sub)transaction. In a MSR local schedule, although a subtransaction g_i and its redo transaction $Redo(g_i)$ execute as independent transactions, they are considered together as a pair. That is, database consistency is preserved by serializing all other transactions executing on the same node with respect to the pair $\{g_i, Redo(g_i)\}$. Such a pair is an instance of cooperative transactions and the history H of interest includes events associated with all transactions, i.e., both committed and aborted transactions. MSR then corresponds to cooperative serializability if all conflicts in this history are considered with respect to two types of cooperative transaction sets: $\{g_j\}$ in case $\{g_j\}$ commits and $\{g_i, Redo(g_i)\}$ in case $\{g_i\}$ aborts.

4.1.4 Quasi Serializability

Quasi Serializability has been proposed in [Du, 1989] as a correctness criterion for maintaining transaction consistency in multidatabases. As mentioned earlier, transactions in these systems are either local, i.e., execute on a single site, or global, i.e., execute on multiple sites. QSR assumes that at most one (sub)transaction of a global transaction executes on a particular site.

In QSR, the correctness of the execution of a set of global and local transactions is based on the notion of a *quasi serial history* which, unlike a serial history, specifies that only global transactions are executed serially. A history is *quasi serial* if (1) all local histories are (conflict) serializable, and (2) there exists a total order of all global transactions g_m and g_n where g_m precedes g_n in the order and all g_m's operations precede g_n's operations in all local histories in which they both appear. A quasi serializable history is equivalent to a quasi serial history.

Let G be the set of global transactions and g_n^s be a (sub)transaction of a global transaction g_n ($g_n \in G$) executing all the operations of g_n on site s.

Let T_s be the set of transactions, both local transactions and global (sub)transactions, executing on site s. $T = (\cup_s T_s)$.

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

Let $\mathcal{R}$ be a binary relation on a set of global transactions G.

Definition 4.14: $\forall\, g_m,\ g_n \ \in\ G, g_m \neq g_n,$

$(g_m\ \mathcal{R}\ g_n)$, if

$$\exists k, g_m^k, g_n^k\ \ (g_m^k\ \mathcal{C}\ g_n^k) \vee \exists p, t_0 = g_m^k, t_p = g_n^k\ \forall i, 1 \leq i \leq p-1\ t_i \in T_k$$
$$\exists ob\ \exists p, q\ ((\mathit{conflict}(p_{t_{i-1}}[ob], q_{t_i}[ob]) \wedge (p_{t_{i-1}}[ob] \rightarrow q_{t_i}[ob])) \wedge$$
$$\exists ob'\ \exists p', q'\ (\mathit{conflict}(p'_{t_i}[ob'], q'_{t_{i+1}}[ob']) \wedge (p'_{t_i}[ob'] \rightarrow q'_{t_{i+1}}[ob'])) \wedge$$
$$(q_{t_i}[ob] \rightarrow p'_{t_i}[ob']))$$

where $\mathcal{C}$ is the binary relation defined in Definition 4.8.

The $\mathcal{C}$ relation captures the fact that two global transactions *directly conflict* in a local history when they invoke conflicting operations on a shared object. Two global transactions might also *indirectly conflict* in a local history even if they do not access any share objects. Indirect conflicts are introduced by other transactions that directly conflict with each other and with the global transactions. These, indirect conflicts between two transactions, particularly those introduced by local transactions, are captured by the second clause of the definition of $\mathcal{R}$. Note that this clause, and consequently $\mathcal{R}$, is not equivalent to the transitive-closure of $\mathcal{C}$ which does not place any restriction on the execution ordering of the conflicting operations, but $\mathcal{R} \subset \mathcal{C}^*$.

DEFINITION 4.15:
H is *quasi serializable* iff

1. $\forall s\ \forall t \in T_s\ \neg(t\ \mathcal{C}^*\ t)$, and
2. $\forall g \in G\ \neg(g\ \mathcal{R}^*\ g)$.

It should be pointed that since the $\mathcal{R}$ relation captures both direct and indirect conflicts between two global transactions in a history, the serializable execution of global transactions is in terms of both direct and indirect conflicts. Indirect conflicts between local transactions induced by conflicts of global transactions that execute on multiple sites are not captured by either clause, the reason being that QSR assumes *no* data dependency across sites.

It is also appropriate to view QSR in terms of cooperative serializability. Specifically, transactions executing in each site s form a cooperative transaction set, with conflict relation $\mathcal{C}$ applied to them. The global transactions form another cooperative transaction set with $\mathcal{R}$ being the conflict relation applied to them.

With respect to the taxonomy of the previous section, for Quasi serializability, (site-based) subsets of the database objects form the consistency units (i.e., objects in each site form a subset) and consistency should hold when a transaction completes. Absolute correctness of transactions' results is expected.

4.1.5 Relationship between Serializability-based Correctness Criteria

Thus far in this section, we have specified four serializability-based correctness criteria using the ACTA formalism and classified them with respect to our taxonomy in Section 2. Here, we will use the formal definitions to relate them to each other. In the next subsection, we will provide a consolidated notion of correctness from which the different serializability criteria can be seen as special cases.

According to predicatewise serializability, each D_k is associated with a $\mathcal{C}_k$ and hence conflicting transactions can be serialized differently with respect to different D_k. This is contrary to serializability which permits only a single system-wide serialization order involving conflicting transactions based on $\mathcal{C}$. However, if D_k is the complete database, then $\mathcal{C} = \mathcal{C}_k$, and consequently, predicatewise serializability is equivalent to serializability.

In the case of cooperative serializability, transactions in different cooperative transaction sets may be related by the $\mathcal{C}_c$ relation (individual transactions not belonging to any cooperative set can be viewed as singleton cooperative transaction sets). Hence, if each cooperative transaction set has just one member, then $\mathcal{C}_c = \mathcal{C}$ and, in this case, cooperative serializability is equivalent to serializability.

In the case of quasi-serializability, there are two distinct conditions under which QSR is equivalent to serializability. These correspond to the situations in which one of the two clauses of the definition of QSR is trivially true: (1) In the absence of global transactions, transactions in T_i are serialized based only on $\mathcal{C}$. (2) In the absence of local transactions, transactions in T_i are serialized based only on $\mathcal{R}$, i.e., here $\mathcal{R} = \mathcal{C}$. Indirect conflicts due to local transactions are not possible whereas indirect conflicts due to global transactions are considered by $\mathcal{C}^*$.

Finally, we would like to point out that these different correctness criteria can be combined and/or adopted within a single database. For instance, it is easy to picture how cooperative serializability can be used in conjunction with predicatewise serializability and how cooperative serializability can be used in conjunction with even quasi-serializability.

For such combinations of correctness criteria, their specification can be derived from the specification of the individual correctness criteria. As an example, let us examine one way that cooperative serializability can be combined with quasi-serializability in order to support a multidatabase system in which component databases allow local transactions to form cooperative groups. In this case, according to cooperative serializability, global (sub)transactions as well as other local transactions that do not belong to a cooperative set, are serialized with respect to all the transactions in the set. The formal definition of this combined criterion is derived from the definition of QSR [Definitions 4.14 and 4.15] by replacing the binary relation $\mathcal{C}$ with a binary relation similar to $\mathcal{C}_c$ defined in the context of CoSR. Specifically:

Let T_i be the set of transactions, both local transactions and global (sub)transactions, executing on site i. $T = (\cup_i T_i)$.

Let T_{c_i} be a set of local cooperative transactions on site i, $T_{c_i} \subseteq T_i$.

Let $\mathcal{C}_{c_i}$ be a binary relation on transactions in T_i.

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

Definition 4.16: $\forall\, t_i,\ t_j,\ t_k\ \in\ T_i, t_i \neq t_j, t_i \neq t_k, t_j \neq t_k\ \forall T_{c_i}$

$(t_i\ \mathcal{C}_{c_i}\ t_j)$, if

$$\exists ob\ \exists p, q\ ((t_i \notin T_{c_i}, t_j \notin T_{c_i}\ (conflict(p_{t_i}[ob], q_{t_j}[ob]) \wedge (p_{t_i}[ob] \rightarrow q_{t_j}[ob]))) \vee$$
$$(t_i \notin T_{c_i}, t_j \in T_{c_i}, t_k \in T_{c_i}\ (conflict(p_{t_i}[ob], q_{t_k}[ob]) \wedge (p_{t_i}[ob] \rightarrow q_{t_k}[ob]))) \vee$$
$$(t_i \in T_{c_i}, t_j \notin T_{c_i}, t_k \in T_{c_i}\ (conflict(p_{t_k}[ob], q_{t_j}[ob]) \wedge (p_{t_k}[ob] \rightarrow q_{t_j}[ob]))))$$

Definition 4.17: H is *local cooperative quasi serializable* iff

1. $\forall i\ \forall t \in T_i\ \neg(t\ \mathcal{C}^*_{c_i}\ t)$, and
2. $\forall g \in G\ \neg(g\ \mathcal{R}^*\ g)$.

4.1.6 Consolidation of the Different Types of Serializability

In this section, we give a single definition of a serializability-based correctness criterion from which the different types of serializability can be derived as special cases.
Let $CoopTr\_Set(T, ob)$ denote the sets of transactions in T where the transactions in each set cooperate in their access to ob. The sets are disjoint with respect to individual objects. Thus, each element of $CoopTr\_Set(T, ob)$ is a cooperative transaction set where each such set is a subset of T.

Definition 4.18:
$CoopTr\_Set(T, ob) = \{TrSet \mid \text{transactions in } TrSet \text{ cooperate over } ob,\ TrSet \subseteq T\}$

For example, let $T = \{t_1,\ t_2,\ t_3,\ t_4\}$ and $DB = \{ob_1, ob_2, ob_3\}$. Let t_1 and t_2 cooperate over ob_3, t_1, t_2, and t_3 cooperate over ob_2, and t_3 and t_4 cooperate over ob_3. Then

$$CoopTr\_Set(T, ob_1) = \{\{t_1\},\ \{t_2\},\ \{t_3\}, \{t_4\}\}.$$
$$CoopTr\_Set(T, ob_2) = \{\{t_1,\ t_2,\ t_3\}, \{t_4\}\}.$$
$$CoopTr\_Set(T, ob_3) = \{\{t_1,\ t_2\},\ \{t_3,\ t_4\}\}.$$

Thus, if no cooperation occurs between transactions, such as in the case of ob_1, each cooperative transaction set is a singleton set. On the other end of the spectrum, if $CoopTr\_Set(T, ob_1) = T$, then we have concurrent process behavior.

Let $\mathcal{C}_c(T, T_c, ob)$ be a binary relation on transactions in T where $T_c \in CoopTr\_Set(T, ob)$. It defines conflict relations that form when transactions access ob given that transactions in T_c cooperate over ob.

Definition 4.19: $\forall\ t_i,\ t_j,\ t_k\ \in\ T, t_i \neq t_j, t_i \neq t_k, t_j \neq t_k$
$(t_i\ \mathcal{C}_c(T, T_c, ob)\ t_j)$ if

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

This is similar to our definition of $\mathcal{C}_c$ except that it considers a specific object ob and a specific set of transactions $T_c \subseteq T$ cooperating over ob.

Let $\mathcal{D}_c(T, OB)$ stand for the union of all $\mathcal{C}_c(T, T_c, ob)$ relations considering all objects ob in the set OB and considering all transactions in the set T.

Definition 4.20:
$(t_i\ \mathcal{D}_c(T, OB)\ t_j)$ iff $\exists ob \in OB\ \exists T_c \in CoopTr\_Set(T, ob)\ (t_i\ \mathcal{C}_c(T, T_c, ob)\ t_j)$.

That is, $\mathcal{D}_c(T, OB)$ contains all the conflicts formed by transactions, after considering cooperation over accesses to objects in OB.

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

Definition 4.21:
H is $Coop\_serializable(OB)$ iff $\forall t \in T\ \neg(t\ \mathcal{D}_c^*(T, OB)\ t)$.
H is *cooperative serializable* iff $Coop\_serializable(DB)$ where DB stands for the set of objects in the database.

Let ADS_Set be the set of atomic data sets in the database, as defined in Section 4.1.2.

Definition 4.22:
H is $Setwise\_serializable(OB)$ if $\forall OB \in ADS\_Set, Coop\_serializable(OB)$ and $\forall ob \in OB, \forall\ s \in CoopTr\_Set(T, ob), |s| = 1$.
H is *Setwise serializable* iff $Setwise\_serializable(DB)$.

Setwise serializability only considers serializability over individual ADS's and does not allow for cooperation. Hence the need for each element of $CoopTr\_Set(T, ob)$ to be a singleton set. In this case, setwise_serializability(DB) corresponds to serwise serializability.

For example, assume $DB = \{ob_1, ob_2, ob_3\}$, $T = \{T_1, T_2, T_3\}$, and $ADS\_Set = \{\{ob_1, ob_2\}, \{ob_3\}\}$. If each cooperative set is a singleton, i.e., there is no cooperation among transactions, cyclic ordering relationships over $\{ob_1, ob_2\}$ will be determined based on $D_c(T, \{ob_1, ob_2\})$ and over $\{ob_3\}$ based $D_c(T, \{ob_3\})$.

In the case that atomic sets in ADS_set are singleton sets, we have independent objects – no consistency constraints exist across objects.

Definition 4.23:
H is *serializable* if H is $Setwise\_serializable(DB)$ and $|ADS\_Set| = 1$.

This follows from the fact that if we have just one ADS, namely the DB, and we allow no cooperation then we get serializability if no cycles occur in the $\mathcal{C}_c^*$ relationship.

In Section 4.1.4 we discussed how one could view QSR in terms of Cooperative serializability with two different conflict relationships $\mathcal{C}$ and $\mathcal{R}$. If we expanded the above definitions to parameterize them with respect to the conflict relationship of interest (instead of just $\mathcal{C}_c$ as assumed above), we can see how QSR can also be consolidated with the above generalized definition of serializability. We do not do it here in the interest of brevity.

These definitions in conjunction with our discussions in the previous section show that it is possible to combine a number of correctness criteria within a single application.

4.2 Transaction Model Dependent and Application Independent Criteria

Transaction model dependent but application independent correctness criteria are typically related to the structure of transactions that conform to a particular model. (Note that specific transaction models may be more suited to specific applications.)

Thus, this section elaborates on the different structure related properties of transactions (See Figure 2). As was mentioned earlier, different transaction models produce different transaction structures where 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 (defined below) on its parent t_p ($t_c\ \mathcal{WD}\ t_p$) and by a parent establishing a commit dependency on its child ($t_p\ \mathcal{CD}\ t_c$). The weak-abort dependency guarantees the abortion of an uncommitted child if its parent aborts whereas the commit dependency prevents a child from committing after its parent has committed.

In [Chrysanthis, 1991] and [Chrysanthis, 1992] we gave axiomatic definitions of different transaction models in terms of dependencies that occur between transactions that conform to a particular model. So we now formally specify some of the dependencies that can occur in addition to the **Commit Dependency**, **Abort Dependency**, and **Begin Dependency** specified in Section 3.2.

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

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.,
$Abort_{t_i} \in H \Rightarrow (\neg(Commit_{t_j} \rightarrow Abort_{t_i}) \Rightarrow Abort_{t_j} \in H)$.

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

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

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

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

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

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

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

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., $Begin_{t_j} \in H \Rightarrow (Commit_{t_i} \in H \Rightarrow (Commit_{t_i} \rightarrow Begin_{t_j}))$.

With respect to the taxonomy, an application that uses an extended transaction model will have correctness requirements related to transactions' structure where these requirements are specified via axioms that express the dependencies that are formed when transactions execute according to the given model.

We first look at some simple examples of structure-related transaction correctness properties. In the transaction model proposed in [Buchmann, 1990, Garcia-Molina, 1991] a parent can commit only if its *vital* children commit, i.e., a parent transaction has an abort dependency on its *vital* children t_v ($t_p\ \mathcal{AD}\ t_v$). Child transactions may also have different dependencies with their parents if the transaction model supports various spawning or coupling modes [Dayal, 1990]. Sibling transactions may also be interrelated in several ways. For example, components of a *saga* [Garcia-Molina, 1987] can be paired according to a compensated-for/compensating relationship [Korth, 1990]. 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 $\mathcal{CAD}$ and strong-commit dependency $\mathcal{SCD}$ [Chrysanthis, 1992]. In a similar fashion, dependencies that occur in the presence of alternative transactions and contingency transactions [Buchmann, 1990] can also be specified [Chrysanthis, 1992].

We now provide a complete example of the structural properties of an open nested transaction model. In an open nested transaction model, component transactions may decide to commit or abort unilaterally.

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

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

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

Suppose some of the components of s are considered vital in that s is allowed to commit only if its *vital* components commit. These components are members of the set *VitalTrs*. We can capture this requirement as follows:

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

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

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

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

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

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

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

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

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

4.3 Transaction and Application Dependent Criteria

We now focus on the required *behavior* of a transaction and hence on the requirements imposed by the application that employs that specific transaction. We distinguish between two types of behavior related properties:

1. Those that relate to constraints on a transaction's access to objects. Some of these are mandated by the concurrency properties of the objects. For instance, as discussed in Section 4.1, serializability demands acyclic $\mathcal{C}$ relationships. Here we will discuss additional access requirements.

2. Those that relate to properties that deal with its other behavioral properties, such as, *when* a transaction can/must begin and *when* it can/must end. Spatial and temporal requirements are related to this type.

Thus, this section corresponds to the data-access related behavior and the temporal behavior of transactions (See Figure 2).

We elaborate upon the first type through an example. Consider a *page* object with the standard *read* and *write* operations, where read and write operations conflict. A read's return-value is dependent on a previous write, whereas a write's return-value is 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 requirement:

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

In this 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. In general, the specification of correct transaction behavior can include 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 [Skarra, 1991].

Let us now turn to other behavioral specifications, for example, those that concern the beginning and termination of transactions. Consider the following simple requirement which states that if *condition* is true then transaction t_j must begin.

$$condition \Rightarrow Begin_{t_j} \in H$$

condition can depend on the occurrence of an event, on the state of the database, and on time. As we will see, the above requirement can be used for the flexible enforcement of consistency, to trigger the propagation of changes, to react to consistency violations, and to notify changes. Thus, the above specification can be considered to be a specification for the automatic triggering of situation-dependent actions, e.g., for expressing the rules that govern the triggering of actions in an active database [Dittrich, 1991].

Suppose *condition* is related to the occurrence of some significant event within a transaction t_i. In this case, the additional structural relationships (for instance, the different *coupling modes* [Dayal, 1990]) between t_i and t_j can be specified via the dependencies discussed in Section 4.2.

If *condition* relates to the state of the database, what we have is related to *spatial consistency* discussed in [Sheth, 1990]. For instance, consider the following *condition*: "One hundred sales have occurred at this store since the master database at the store's headquarters was last updated." If *condition* relates to time, for instance, if *condition* is "$time > 8pm$", we have a *temporal consistency* requirement.

Now let us consider situations where *constraints* are placed on the beginning of transactions. For example, a transaction t_i to compute daily interest can start after midnight but only after the day's withdrawals and deposits have been reflected in the account (say by a transaction t_j). This can be specified as

$$(ts(Begin_{t_i}) \geq 12am) \wedge (t_i \ \mathcal{BCD} \ t_j).$$

This is an example where a transaction has a time-based start-dependency as well as a begin-on-commit dependency on another transaction.

Let us consider another example. If a deposit is made by time x then the transaction that reflects it in the account should not be started until time y. This is specified by

$$(Commit_{t_i} \in H \wedge (ts(Commit_{t_i}) < x)) \Rightarrow (Begin_{t_j} \in H \Rightarrow (ts(Begin_{t_j}) > y)).$$

Through several examples, we now consider requirements and constraints associated with the termination of transactions.

Sometimes, we may want to specify that some specific change of state (by one transaction) triggers [Dayal, 1990] another transaction (that perhaps fixes the inconsistency resulting from the first transaction). Clearly, this type of constraint is related to deferred consistency restoration. This can occur, for example, if we had two versions of a database, one which was complete and another (at perhaps a different site) which only contained data required at that site. The two are not required to be consistent at all times but changes done to the complete database are required to percolate to the other within a specified delay. If the changes should be reflected within d units of time, we have the following "temporal commit dependency":

$$(Commit_{t_i} \in H \wedge (ts(Commit_{t_i}) = t)) \Rightarrow Commit_{t_j} \in H^{t+d}.$$

This says that if t_i commits at time t, t_j should commit *by* time $t + d$. For another example, consider the following:

$$(temperature \geq threshold \wedge time = t) \Rightarrow Commit_{t_j} \in H^{t+function(temperature)}$$

Here t_j could be a transaction that opens a valve to pass more coolant into the reactor whose temperature is above *threshold*. The length of time available to complete this transaction is a function of the current temperature. This is a form of triggered transaction but with specific time constraints imposed on its completion [Korth, 1990b]. Such time constrained activities occur in real-time databases [Ramamritham, 1993a].

In some situations, it may be desirable to specify an interval $[l, u]$ such that t_j does not commit before l (the lower bound) but definitely commits before u (the upper bound). For example, consider deposits into a bank account. During the day, if a deposit is made before 3pm, it is just "logged" into a file but is reflected in the appropriate account between 10pm and 4am that night. Such constraints take the form

$$(Commit_{t_i} \in H \wedge (ts(Commit_{t_i}) < t)) \Rightarrow Commit_{t_j} \in H^{[l,u]}$$

The above conditions imposed on the initiation and termination of transactions can be viewed as generalizations of the preconditions and postconditions associated with specific transactions [Korth, 1988b].

For a final example of behavior related specifications, consider the situation in which it may be desirable to prevent a transaction t_i from aborting after a time t. This corresponds to the assumption that a transaction is implicitly committed if it has not aborted by a certain time [Rusinkiewicz, 1990]. For example, no bets can be canceled after a race is started and a lottery ticket cannot be refunded after a given time:

$$Abort_{t_i} \notin H^t \Rightarrow Commit_{t_i} \in H^{t+1}.$$

5 Conclusions

In this paper, we have examined different types of correctness criteria and have attempted to provide a taxonomy with respect to *database consistency requirements* and *transaction correctness properties*. Given space limitations, we could examine, in detail, only a subset of the proposals that have been made to capture the correctness properties applicable to extended transaction models as well as those demanded by the newer database applications.

We have approached the problem of categorizing the different proposals by formally specifying them using the framework of ACTA. This allows us to clearly see where one proposal differs from another and what its relationship with serializability is.

It is important to point out that even though we have used the word transaction in this paper to refer to computations that have transaction-like properties, as we have seen, such computations can have inter-related components. In the literature, such computations have also been termed *activity* models or *workflow* models. Dependencies such as the ones defined in Section 4.2 as well as more general requirements such as the ones exemplified in Section 4.3 allow us to capture relationships between components of these activities or workflows and thus allow us to specify the correctness requirements of these computations.

We believe this taxonomy to be a good starting point in our endeavor to classify proposed correctness criteria and to compare and contrast them. It can be viewed as a common framework with respect to which one can study where a new correctness criterion fits and how it relates to existing criteria. In this regard, we expect the taxonomy to evolve as better understanding is gained about the correctness needs of emerging database applications.

Let us now examine some of the other implications of this work. In a recent survey [Breitbart, 1992], the authors present a hierarchy of serializable schedules for multi-databases, each placing different types of restrictions on transaction management within individual databases and the global database. In the conclusion of that paper they point out the need to consider alternatives to the "standard" notions of consistency. One of the intended contributions of our work has been to review various alternatives that have been proposed and to place them in perspective. In multi-database systems, the specifications of database consistency and transaction correctness can be viewed as requirements on the coordinator of the blackboxes [Breitbart, 1990] that control individual databases.

In terms of future work, we would like to see the reasoning capabilities of the formalisms, such as ACTA, being used to study the properties of mechanisms, such as in [Sheth, 1991], for maintaining relaxed correctness properties of interdependent data. In the same context, it will be useful to investigate ways in which the formal primitives themselves can be used as part of these mechanisms [Rusinkiewicz, 1991, Sheth, 1992]. This is in line with the work on the ConTract Model [Wachter, 1991] and CACS [Stemple, 1992]. Here, transactions are made up of multiple steps, with explicit dependency relationships specified between the steps. The system ensures that such dependencies hold when the steps execute.

Acknowledgements

The authors thank Alex Buchmann for lively discussions about notions of consistencies and correctness, and Nandit Soparkar, Amit Sheth, Greg Speegle and the anonymous referees for their comments on previous versions of this paper.

References

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

Barghouti, N. S. and G.E. Kaiser. (1991) Concurrency Control in Advanced Database Applications. *ACM Computing Surveys*, 23(3), pages 269-317.

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

Bernstein P. A., V. Hadzilacos, and N. Goodman. (1987) *Concurrency Control and Recovery in Database Systems.* Addison-Wesley.

Breitbart Y. et al. (1990) Final Report of the Workshop on Multidatabases and Semantic Interoperability.

Breitbart Y., D. Georgakopoulos, M. Rusinkiewicz and A. Silberschatz. (1991) On Rigorous Transaction Scheduling, *IEEE Transactions on Software Engineering*, 17(9):954–960.

Breitbart Y., H. Garcia-Molina, and A. Silberschatz. (1992) Overview of Multidatabase Transaction Management. *VLDB Journal* Vol.1, No.2.

Buchmann. A, M. Hornick, E. Markatos, and C. Chronaki. (1990) 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.

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

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

Chrysanthis P. K. and K. Ramamritham. (1991) A Formalism for Extended Transaction Models. In *Proceedings of the seventeenth International Conference on Very Large Databases*, pages 103–112.

Chrysanthis P. K. and K. Ramamritham. (1992) ACTA: The SAGA Continues. In A. K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications.* Morgan Kaufmann, pages 349-397.

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

Dittrich K. R. and U. Dayal. (1991) Active Database Systems (Tutorial Notes). In *The Seventeenth International Conference on Very Large Databases.*

Du W. and A. K. Elmagarmid. (1989) 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.

Elmagarmid A. (Editor). (1992) *Database Transaction Models for Advanced Applications.* Morgan Kaufmann, 1992.

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

Farrag A. and M. T. Ozsu. (1989) Using semantic knowledge of transactions to increase concurrency. In *ACM Transactions of Database Systems*, 4(14):503–525.

Fekete A., N. Lynch and W. Weihl. (1990) A serialization graph construction for nested transactions. In *Proceedings of the 9th ACM Symposium on Principles of Database Systems.*

Garcia-Molina H. (1983) Using semantic knowledge for transaction processing in a distributed database. *ACM Transactions on Database Systems*, 8(2):186–213.

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

Garcia-Molina H. and Kenneth Salem. (1987) SAGAS. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 249–259.

Gray J. N., R. A. Lorie, G. R. Putzulo, and I. L. Traiger. (1975) 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.

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

Herlihy M. P. and J. M. Wing. (1987) Axioms for Concurrent Objects. In *Proceedings of the Fourteenth ACM Symposium on Principles of Programming Languages*, pages 13–26.

Hou. W, G. Ozsoyoglu, and B. K. Taneja. (1989) Processing Aggregate Relational Queries with Hard Time Constraints. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 68–67.

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

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

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

Korth H. F., N. Soparkar, and A. Silberschatz. (1990b) Triggered Real-Time Databases with Consistency Constraints. In *Proceedings of the Sixteenth International Conference on Very Large Databases*, pages 71–82

Korth H. F. and G. Speegle. (1990c) Encapsulation of Transaction Management in Object Databases. In *Proceedings of the OOPSLA/ECOOP'90 Workshop on Transactions and Objects*, pages 27–32.

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

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

Martin B E. and C. Pedersen. (1992) Long-Lived Concurrent Activities. In T. Öszu, U. Dayal and P. Valduries, (eds), *Distributed Object Management*. Morgan Kaufmann.

Mehrotra S., R. Rastogi, H. Korth, and A. Silberschatz. (1991) Non-serializable Executions in Heterogeneous Distributed Database Systems. In *Proceedings of the First International Conference on Parallel and Distributed Information Systems*.

Mehrotra S., R. Rastogi, Y. Breitbart, H. Korth, and A. Silberschatz. (1992) Ensuring transaction atomicity in multidatabase systems. In*Proceedings of the 11th Symposium on Principles of Database Systems,* pages 164–175.

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

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

O'Neil P. E. (1986) The Escrow Transactional Method. *ACM Transactions on Database Systems*, 11(4):405–430.

Pu C., G. Kaiser, and N. Hutchinson. (1988) Split-Transactions for Open-Ended activities. In *Proceedings of the Fourteenth International Conference on Very Large Databases*, pages 26–37.

Pu C. and A. Leff. (1991) Replica Control in Distributed Systems: An Asynchronous Approach. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 377–386.

Ramamritham K. (1993a) Real-Time Databases. *International Journal of Distributed and Parallel Databases* Volume 1, Number 2, pp. 199- 226.

Ramamritham K. and C. Pu. (1993b) A Formal Characterization of Epsilon Serializability. *IEEE Transactions on Knowledge and Data Engineering*, (to appear).

Rusinkiewicz M., A. Elmagarmid, Y. Leu, and W. Litwin. (1990) Extending the Transaction Model to Capture more meaning. *ACM Sigmod Record*, 19(1):3–7.

Rusinkiewicz M., A. Sheth, and G. Karabatis. (1991) Specification of Dependencies for the Management of Interdependent Data. *IEEE Computer*, 12(12):46–54.

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

Sheth A. and M. Rusinkiewicz. (1990) Management of Interdependent Data: Specifying Dependency and Consistency Requirements. In *Proceedings of the Workshop on the Management of Replicated Data*, pages 133–136.

Sheth. A, Y. Leu, and A. Elmagarmid. (1991) Maintaining consistency of interdependent data in multidatabase systems. Technical Report CSD-TR-91-016, Computer Science Department, Purdue University.

Sheth A., M. Rusinkiewicz, and G. Karabatis. (1992) Polytransactions: A Mechanism for Management of Interdependent Data. *book chapter in Transaction Models for Advanced Database Applications, Ed. A. Elmagarmid,* Morgan-Kaufmann.

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

Stemple, D. W. and R. Morrison. (1992) Specifying flexible concurrency control schemes: An abstract operational approach. In *Annual Australian Computer Science Conference.*

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

Weikum, G. and H.-J. Schek. Architectural Issues of Transaction Management in Layered Systems. *Proceedings of the 10th Conference on Very Large Databases* pp 454-465, 1984.

Yannakakis M. (1984) Serializability by Locking. *Journal of the ACM*, 31(2):227–244.