In Search of Acceptability Criteria: Database Consistency Requirements and Transaction Correctness Properties

K. Ramamritham and P. Chrysanthis

COINS Technical Report 91-92
December 1991

In Search of Acceptability Criteria: Database Consistency Requirements and Transaction Correctness Properties [1]

Krithi Ramamritham

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

Panos K. Chrysanthis

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

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 applied 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.

[1]This material is based upon work supported by the National Science Foundation under grant IRI-9109210.

Contents

1 Introduction **1**

2 A Taxonomy of Correctness Criteria **3**
2.1 Database Consistency Requirements 3
2.2 Transaction Correctness Properties 6

3 A Quick Introduction to the ACTA Formalism **8**
3.1 Significant Events Associated with Transactions 8
3.2 History, Projection of the History, and Constraints on Event Occurrences . 9
3.3 Objects, Operations, and Conflicts 10

4 Specification and Categorization of Correctness Criteria **11**
4.1 Transaction and Application Independent Criteria 12
4.1.1 Serializability 12
4.1.2 Predicatewise Serializability 13
4.1.3 Cooperative Serializability 14
4.1.4 Quasi Serializability 15
4.2 Transaction Model Dependent and Application Independent Criteria 16
4.3 Transaction and Application Dependent Criteria 18

5 Conclusions **22**

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. That is, they deal, for example, with the results of transactions and the interactions between transactions. Serializability [14] 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, or *acceptability criteria*, that view these two types of requirements independently. The search for more flexible correctness requirements is also motivated by the introduction of new transaction models that extend the traditional atomic transaction model. (See [13] 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.

Proposed correctness criteria range from the standard serializability notion to eventual consistency [40]. Quasi-serializability [12], predicate-wise serializability [25], 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.

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 [7, 8] 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

We study 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.

- *Consistency Unit:*
  - *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.

    Example: Traditional serializability (SR) applied to atomic transactions [4].
  - *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. 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: Set-wise serializability (SSR) applied to *compound transactions* [39] and Predicate-wise serializability (PSR) applied to cooperative transactions [22].
    * *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) [12] and its generalization [30] applied to distributed transactions.

- *Individual Objects*:

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

  Example: Linearizability [20] applied to objects accessed by concurrent *processes.*

- *Consistency Maintenance:*

  This is related to the issues of *when* a consistency requirement is expected to hold and *how* consistency is restored if it does not hold.

  - *When is a consistency requirement expected to hold?*

    * *At activity boundaries*: (An activity denotes a unit of work)

      · *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 [15] and multilevel atomicity [27]

      · *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 [25].

* *At specific points of time*:

  Consistency is required only at/after specific points in time. This is an example of *temporal consistency* [40].

  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.

* *At 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.

  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 [40].

– *If a consistency requirement does not hold at a point it is supposed to, how is it restored*?

* *Restored immediately*:

  This applies when the consistency requirement is required to hold at activity boundaries. The activity is allowed to complete only if the requirement holds, i.e., completion is delayed until consistency holds.

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

* *Restored in a deferred manner*:

  This typically applies when consistency is expected in certain states or at certain times. When it is applied to consistency that is expected at an activity boundary, the activity is allowed to complete and restoration is begun subsequently.

  · *Eventually*:

    Consistency between objects must be restored eventually.

Example: If mutual consistency is required between two objects and one is changed by a transaction, another can be *triggered* to make changes in the other object.

- *By a certain time*:

  A deadline may be imposed on the time by which consistency is restored.

  Example: In real-time systems, the state of the controlled environment should be reflected in the internal state of the controlling system within a certain time so that appropriate and timely control can be exercised.

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 three criteria:

- *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) [35] applied to Epsilon-transactions, approximate query processing [21].

- *Correctness of transaction structure*:
  Correctness depends on the (structural) relationship between transactions. This typically translates into *commit, abort, begin*, and other types of dependencies [8] between transactions.

  Example: Sagas [17], multi-level serializability [26].

- *Correctness of transaction behavior*:

  - *Data access related behavior*:
    Transactions are required to perform operations on objects in a certain manner to be considered correct.

    Example: patterns [41].

  - *Temporal behavior*:
    Transactions have start time and completion time (deadline) constraints.

    Example: Transactions in real-time systems.

The taxonomy just presented shows how the various weakened versions of serializability can be viewed from the perspectives of database consistency and transaction correctness. We revisit these notions in Section 4.1 where they 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 [8]. 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 [31], *Split*, found in the split transaction model [34], 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 $<_t$ where $<_t$ 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[2]. Clearly, $ts(\beta)$ will be larger than $ts(\alpha)$ if α appears earlier in the partial order. 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* [4] 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.

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.

[2]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.

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 $(t_j \; CD \; t_i)$. In general, $((Commit_{t_i} \in H) \Rightarrow condition)$ specifies that *condition* should hold for t_i 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 $(t_j \; AD \; t_i)$. 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 history with respect to the operation invocations 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 functional composition; $H \circ p$ appends p to history H.) Thus, two operations conflict if their effects on the state of an object or their return values are not independent of their execution order. 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 [8].)

4.1 Transaction and Application Independent Criteria

In general, transaction and application independent correctness criteria are extensions to serializability. In this section, we specify some of these extensions using the formalism described in the previous section.

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.,

$(t_i \mathcal{R}^* t_k)$ 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 preserving serializability*, is the well-accepted criterion for concurrency control.

DEFINITION 4.8: Let $\mathcal{C}$ be a binary relation on T, $t_i, t_j \in T$.

$(t_i \mathcal{C} t_j), t_i \neq t_j$ 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: A set of transactions T is *serializable* iff $\forall t \in T, \neg(t \mathcal{C}^* t)$.

To illustrate the practical implications of these definitions, let us consider the case where all operations perform in-place updates. 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 [8] 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.) The

commit order induced by the C relation corresponds to the serialization order. By requiring that there be no cycles in the C relation, the above definition states that the commit order, and hence the serialization order, must be acyclic.

With respect to the taxonomy of Section 2, for serializability, the consistency unit is the complete database, consistency is required at transaction boundaries, and immediate consistency restoration is required. 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 e.g. see [2, 19, 33]) can be used to define the *conflict* relationship between operations. Furthermore, different *degrees* of consistency [18] 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 [18] and with ESR [35] (see [37] for a formal characterization of ESR). The latter is the case with abstract serializability – used in the context of multi-level transactions [3, 32, 28, 2].

4.1.2 Predicatewise Serializability

Predicatewise serializability has been proposed in [22, 25] 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 4.10: Let $C = C_1 \wedge C_2 \ldots \wedge C_n$ be the database consistency constraint. Suppose the disjunctive clause C_k relates to objects in $D_k \subseteq DB$, where DB is the database.

DEFINITION 4.11: Let C_k be a binary relation on transactions in T, t_i t_j, $\in T$.

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

DEFINITION 4.12: A set of transactions T is *predicatewise serializable* iff

$$\forall t \in T \; \forall D_k \; 1 \leq k \leq n \; \neg(tC_k^*t)$$

In [39], each D_k is said to be an *atomic data set.* With respect to the taxonomy, for predicatewise serializability, an atomic data set [39] forms a consistency unit, consistency is required at transaction boundaries, and immediate consistency restoration is required. Absolute correctness of transaction results is expected. Compound transactions [39] behave according to the predicatewise serializability correctness criterion.

4.1.3 Cooperative Serializability

We define *cooperative serializability* (CSR) with respect to a set of transactions which maintain some consistency properties. 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 4.13: T_c be a set of cooperative transactions, $T_c \subseteq T$. Let C_c be a binary relation on T, t_i, t_j, $t_k \in T$.

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

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

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 4.14: A set of transactions T is *cooperative serializable* iff

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

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, and immediate consistency restoration is required. Absolute correctness of transaction results is expected. The correctness requirement expressed informally in [29] corresponds to CSR.

4.1.4 Quasi Serializability

Quasi Serializability has been proposed in [12] 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_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.

DEFINITION 4.15: Let T_i be the set of transactions, both local and global, executing on node i.

DEFINITION 4.16: Let $\mathcal{R}$ be a binary relation on a set of global transactions G, and $g_m, g_n \in G$.

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

$\exists k \ \exists g_m, g_n \in T_k \ \exists ob \ \exists p, q \ (conflict(p_{g_m}[ob], q_{g_n}[ob]) \wedge (p_{g_m}[ob] \rightarrow q_{g_n}[ob]))$

DEFINITION 4.17: A set of local and global transactions $T = (\cup_i T_i)$ $(G \subseteq T)$ is *quasi serializable* if

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

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), consistency is required when a transaction completes, and immediate consistency restoration is required. Absolute correctness of transactions' results is expected. Quasi serializability has been proposed in the context of distributed (multi-database) transactions.

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.) 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.

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.,

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

Let us look at further examples of structure-related transaction correctness properties. In the transaction model proposed in [6, 16] 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 [10]. Sibling transactions may also be interrelated in several ways. For example, components of a *saga* [17] can be paired according to a compensated-for/compensating relationship [23]. 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 $\mathcal{SCD}$ [9]. In a similar fashion, dependencies that occur in the presence of alternative transactions and contingency transactions [6] can also be specified.

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 – beyond those mandated by the concurrency properties of the objects.

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.

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 uncommitted 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 [41].

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 [11].

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* [10]) 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 [40]. 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

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

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 [10] 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 [24].

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 [25].

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 [38]. For example, no bets can be canceled after a race is started and a lottery ticket cannot be refunded after a given time.

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

Other examples of behavior related requirements appear in [36].

5 Conclusions

In this paper, we have examined different types of acceptability 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.

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 the context of a multi-database system, the specifications of database consistency and transaction correctness can be viewed as requirements on the coordinator of the blackboxes [5] that control individual databases. We would like to apply the reasoning capabilities of the ACTA formalism to study the properties of mechanisms, such as in [1], for maintaining relaxed correctness properties of interdependent data. In the context of active databases, we can see how the semantics of the rules that govern the triggering of actions can be formally specified. In addition, the relationships between the triggering action and the triggered action can also be expressed precisely using dependencies. Just as we were able to reason about extended transactions using ACTA [8], we see the formalization of different aspects of active databases as the starting point for addressing issues, such as, reasoning about the consequences of rule firings, changes to rules, and coupling modes.

Acknowledgments

The authors thank Alex Buchmann for lively discussions about notions of consistencies and correctness.

References

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

[2] 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.

[3] 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.

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

[5] Breitbart Y. et al. Final Report of the Workshop on Multidatabases and Semantic Interoperability. Technical report, Tulsa, OK, November 1990.

[6] 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.

[7] 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.

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

[9] Chrysanthis, P. K. and Ramamritham, K. ACTA: The SAGA Continues. In Elmagarmid, A. K., editor, *Database Transaction Models for Advanced Applications.* Morgan Kaufmann, 1991.

[10] 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.

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

[12] 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.

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

[14] 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.

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

[16] 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.

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

[18] 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.

[19] 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.

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

[21] Hou, W., Ozsoyoglu, G., and Taneja, B. K. Processing Aggregate Relational Queries with Hard Time Constraints. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, May 1989.

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

[23] 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.

[24] 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.

[25] 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.

[26] Korth, H. F. and Speegle, G. Encapsulation of Transaction Management in Object Databases. In *Proceedings of the OOPSLA/ECOOP'90 Workshop on Transactions and Objects*, pages 27–32, Ottawa, Canada, October 1990.

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

[28] 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.

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

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

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

[32] 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.

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

[34] 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.

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

[36] Ramamritham, K., Chrysanthis, P. K., and Buchmann, A. Degrees of Consistency – Specification and Maintenance. Technical report, Computer Science Department, University of Massachusetts, 1991.

[37] Ramamritham, K. and Pu, C. A Formal Characterization of Epsilon Serializability. In *(submitted for publication)*, November 1991.

[38] Rusinkiewicz, M., Elmagarmid, A., Leu, Y., and Litwin, W. Extending the Transaction Model to Capture more meaning. *ACM Sigmod Record*, 19(1), January 1990.

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

[40] Sheth, A. and Rusinkiewicz, M. Management of Interdependent Data: Specifying Dependency and Consistency Requirements. In *Proceedings of the Workshop on the Management of Replicated Data*, pages 133–136, Huston, Texas, November 1990.

[41] 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.