Conditional Conflict Serializability—An Application Oriented Correctness Criterion

Ole J. Anfindsen
Telenor R&D, Norway

Serializability is too strict a correctness criterion for several application domains, in particular where support for long-lasting transactions is required. This paper describes a generalized version of serializability called conditional conflict serializability (CCSR), which is built on a customized notion of conflict rather than the standard commutativity-based one. The actual customization of conflicts is carried out by applications that associate parameters with their read and write operations. The semantics of such parameters are user-defined, and can be chosen to suit various needs. CCSR can be enforced by means of two phase locking with parameterized locks. Transaction histories that are strict or rigorous modulo CCSR are defined, showing that a CCSR scheduler need not rely on compensating actions for recovery.

Standard transaction management ensures the ACID properties for transactions (atomicity, consistency, isolation, and durability; Härder & Reuter 1983, Gray & Reuter 1993). Of primary interest here is the isolation property, which corresponds to the correctness criterion serializability. It is generally acknowledged in the database research community that serializability is unsuitable for several application domains (Elmagarmid 1992, Kaiser 1995, Ramamritham & Chrysanthis 1996), in particular where transaction duration is long relative to response time requirements. This paper presents conditional conflict serializability (CCSR), a correctness criterion capable of supporting serializability, dirty reads, and almost any degree of concurrency in between those two extremes. This makes CCSR suitable for almost any application domain where read-write and write-read conflicts need to be handled in a more lenient way than with unconditional conflict serializability. In particular, CCSR is appropriate for applications that need long-lasting transactions.

An important motivation behind CCSR is the need for a correctness criterion that is both general and simple. There is a need for a framework that offers application programmers easy-to-use customization of concurrency control, without affecting programmers who need their applications to be fully isolated from others. Another major motivation behind CCSR has been to enable arbitrary levels of concurrency without forcing programmers to perform recovery by means of compensating actions.

The larger context for the results presented in this paper is Apotram (which stands for application-oriented transaction model; Anfindsen 1997). In particular, it should be noted that while CCSR has nothing to offer in terms of handling write-write conflicts, it is part of a framework where this is taken care of. The Apotram mechanism for handling write-write conflicts is called nested databases (Anfindsen 1996; 1997, 45-52), and the corresponding correctness criterion is called nested conflict serializability (NCSR). NCSR and CCSR can be combined into a single correctness criterion.

Related Work

The ideas set forth in this paper are related to those of Farrag & Özsu (1989) and their correctness criterion Relatively Consistent Schedules (RELC), which is a
generalization of the correctness criteria of Garcia-Molina (1983) and Lynch (1983). All these references, just like this paper, are aimed at allowing greater concurrency than is possible with serializability. A detailed analysis of the differences and similarities of RELC and CCSR can be found in (Anfinsen 1997, 93–96), the main conclusion of which is that a CCSR scheduler can produce all RELC histories except those that are inherently unrecoverable. This is a reflection of the different approaches taken to recovery; while (Garcia-Molina, 1983; Farrag & Özsu, 1989; Lynch, 1983) all rely on compensating actions, the Apotram/CCSR approach is based on the premise that only as a last resort should one be willing to compromise the commitment atomicity of transactions, because we then also sacrifice their recoverability. In other words, this paper, like the just mentioned references, aims at increased concurrency, but without giving up the benefits of system-controlled recovery.

Ammann et al (1997) aim at increased concurrency by means of semantic-based decomposition of transactions. The focus of their paper is on formal methods used to obtain correct decompositions. The model of Ammann et al is closely related to, but more general than, the one from Farrag & Özsu (1989). Both models rely on recovery by means of compensating actions.

Shasha et al (1995) investigate how to increase concurrency in database systems by chopping up transactions based on information supplied by a database system user. Their goal is to ensure serializability “without paying for it.” According to Ammann et al (1997, 240), the approach of Shasha et al is less general than their own, “Since they do not use any semantic knowledge.”

Kirsch et al (1994a) introduce Database Conversations, which is “an application-independent, tight framework for jointly modifying common data.” In their model each data unit has a binary conversation flag associated with it, and the two make up an indivisible unit; “the information whether a data unit is uncertain or not, is stored explicitly with the data item itself and not implicitly in some transaction semantics.” If the conversation flag is set, it means that a conversation context has been created for the data item in question. Conversation contexts are used for coordinating updates to the data item from multiple transactions, allowing one to start from the old value, step over a number of intermediate values, and eventually arrive at the final value. During this, the data item can be read by other transactions, which will then see its old value but will also be told by the conversation flag that this value is unreliable.

The conversation flags of (Kirsch et al., 1994a) correspond somewhat to Apotram’s notion of access mode parameters. Both are used to tell other transactions something about the status of data, both are examples of the use of metadata, and both are based on the notion of uncommitted data being uncertain. But while conversation flags have only two values, access mode parameters can have an arbitrary number. Thus, this paper, like (ibid), is built on the observation that uncommitted data is unreliable, and that transactions therefore should communicate with each other on the status of such data. The uncertainty associated with uncommitted data is further elaborated in (Kirsche et al., 1994b). While the language for intertransaction communication in (Kirsche et al., 1994a) is based on dynamically created conversation contexts, CCSR is based on attaching parameters to read and write operations.

Muth (1997) “propose a transaction model for multidatabase systems - the heterogeneous 3-level-transaction model - which can be parameterized to support different compromises between global serializability and local autonomy, and fully supports recovery of global transactions.” (ibid, 358, emphasis in original). This model is based on multi-level transactions (Weikum, 1991). Just as Apotram/CCSR, the heterogeneous 3-level-transaction model regards the commutativity-based notion of conflict as being too restrictive in many cases. Four different kinds of information is exploited in order to provide “different alternatives for conflict definitions between local transactions and global subtransactions” (Muth 1997, 359). These are system architecture, semantics of actions, actions allowed, and consistency constraints (ibid, 371-372). The importance of this is commented as follows: “This is the key of our work, as it allows tailoring the model to specific application needs.” (ibid, 359). The heterogeneous 3-level-transaction model is clearly more complicated than Apotram/CCSR, but Muth remarks that “most of the complexity is in the theoretical foundation” and that “from an application point of view” using his model will be “rather simple” (ibid, 396). Another difference from Apotram/CCSR is that Muth relies on recovery by means of “inverse actions,” i.e., compensation.

Agrawal et al (1993) introduce three new correctness criteria; consistency, orderability, and strong orderability. Consistency is based solely on database consistency assertions specified by the users, and the users must make sure assertions are chosen in such a way that “nothing unexpected happens” (ibid, 469). Orderability and strong orderability are, just like CCSR, based on notions of equivalence with serial histories, and are generalizations of view and conflict serializability, respectively. Agrawal et al sum up the main differences between their own model and traditional models, of which Apotram/CCSR is an example, as follows (ibid, 482-483): “First, our model assumes abstract data types, which support semantically rich operations, whereas the traditional model is limited to read and write operations. Second, in our model users specify predicates on the database through consistency assertions before every operation.”

Korth & Speegle (1994) introduce a transaction model called nested transactions with predicates and versions, or NT/ PV, which generalizes conflict serializability. In this model each update operation generates a new version of the data item being updated, and the model relies on the use of explicit predicates to form pre- and postconditions for transactions.
These two features are also NT/PV’s most obvious differences from the approach taken with Apotram and CCSR.

At a more abstract level, this paper is related to the philosophy behind the Customizable Transaction Processing Engine (CTPE) of Daynes et al (1997) and the Transaction Specification and Management Environment (TSME) of Georgakopoulos et al (1993, 1994). In particular, Apotram was influenced by the basic idea of TSME that transaction management should be tailored to meet the specific needs of different applications.

### Conditional Conflict Serializability

One idea behind CCSR is to depart from a purely commutativity-based notion of conflict. While write operations should still be considered mutually conflicting, write-read and read-write conflicts should be made conditional; hence conditional conflict serializability. This is achieved by means of parameterized read and write modes, and correspondingly parameterized read and write locks. Let R(A) and W(B) denote a parameterized read and write mode respectively, where A and B denote sets of parameters drawn from some parameter domain D, i.e. A, B ⊆ D. Then R(A) and W(B) are compatible iff B ⊆ A, otherwise R(A) and W(B) conflict (iff is shorthand notation for if and only if). This is illustrated by the compatibility matrix of Table 1.

In standard transaction theory we say that two histories are conflict equivalent if

1. They contain the same transactions and the same operations; and
2. Conflicting operations of non-aborted transactions are ordered the same way in both histories.

Identical terms can be used to define conditional conflict equivalence, provided “conflicting operations” are performed by different transactions on the same data item and understood as any pair of write operations or any pair of operations R(A) and W(B) such that B is not a subset of A. Informally, “conflicting operations” should now be understood as operations that conflict even when their access mode parameters have been taken into consideration.

**Definition 1**: A history is defined as conditional conflict serializable (CCSR) iff it is conditional conflict equivalent with a serial history.

It should be emphasized here that the parameter domain D is user defined and could contain few or many parameter values; whatever suits the applications at hand. In any case, all application programmers that need to use parameterized reads or writes, must have a common understanding of the semantics of the parameters they use. However, programmers need not even be aware of, much less understand the semantics of, parameters that are defined in D, but which their applications make no use of. Transaction managers that support CCSR need not understand anything about the semantics of read and write parameters at all.

**CCSR and The Serializability Theorem**

The fundamental theorem of serializability theory says that “a history H is serializable iff SG(H) is acyclic” (Bernstein et al 1987, 33). This theorem can be generalized to cover CCSR. Consider a serialization graph for a transaction history H, denoted SG(H); it is a directed graph whose nodes are the committed transactions of H, and it has an edge between all pairs of nodes such that the two transactions in question have issued conflicting operations. A verbatim copy of this statement can be used to define a conditional conflict serialization graph (CCSG), provided the term “conflicting operations” is understood in the conditional sense. In other words, CCSG(H) has an edge (T_i, T_j) iff T_i has issued an operation and later T_j has issued another operation that is conflicting with the earlier one of T_i, even when the access mode parameters of the two operations are taken into consideration.

**Theorem 1** (the Generalized Serializability Theorem): A history H is CCSR iff CCSG(H) is acyclic.

Bernstein et al (1987) use a proof for the Serializability Theorem that hinges on the ordering of conflicting operations. Since they made no assumption about a specific definition of conflict, their proof will apply to different versions of conflict serializability that use different definitions of conflict. In particular, their proof applies to CCSR.

### Parameterized Access Modes

The basic idea of parameterized read and write modes is that users should be able to specify when reading and writing should be incompatible. In other words, the standard notion that read and write modes are mutually incompatible is reduced to a default which transactions may override by using parameters.

Let D denote the domain of access mode parameters, and A, B arbitrary subsets of D. Parameterized read and write
modes will be denoted as R(A) and W(B) respectively. R(A)
and W(B) are compatible iff B \[ \cap \] A. For example, if good, bad,
\textit{incomplete} D then it would be the case that:

- R(good) and W(good) are compatible,
- R(good) and W(bad) are incompatible,
- R(good, bad) and W(bad) are compatible,
- R(bad) and W(bad, incomplete) are incompatible.

Users may specify the parameters that will be used
during read and write access. Non-parameterized read
and write modes can still be denoted as R and W but should, in the
interest of generality, be thought of as R(\emptyset) and W(*)
respectively, where \emptyset denotes the empty set and D \( \not\in \) * (i.e.,
\( * \) denotes an arbitrary superset of D). Thus, according to
the rule that R(A) and W(B) are compatible iff B \[ \cap \] A, R(\emptyset) will
be incompatible with all write modes and W(*) will be
incompatible with all read modes. Also, it follows from this
notational convention that there is no such thing as a write
mode that is compatible with every read mode, since W does
not mean W(\emptyset) but, rather W(*). R(D) denotes the read mode
that is compatible with write mode W(B), for any non-empty
B \[ \cap \] D, but incompatible with W.

A parameterized write mode indicates willingness to
share information with readers. The idea is that a
parameterized writer indicates to other transactions the
reliability class of its data. Analogously, the use of parameterized
read modes indicates willingness to read data that belongs to
parameterized writers whose data belongs to a certain reliability
class. The new access mode compatibility matrix is an extension
of the traditional one, as shown in Table 2.

**Implementation of Parameterized Access Modes**

Parameterized access modes can be implemented by
correspondingly parameterized lock modes (assuming that
locking is the chosen concurrency control method). There will
be one lock mode per access mode, and intent locks in
addition; IR, IW, and RIW (also known as IS, IX, and SIX;
Gray & Reuter 1993, 406-411). RIW locks need both a read
and a write parameter (in general), and is thus denoted

R(A)IW(B). The semantics of this lock can be paraphrased as
read with parameter set A and intention to write with
parameter set B. Two locks R(A)IW(B1) and R(A)IW(B2)
will be mutually compatible iff B1 \[ \cap \] A2 and B2 \[ \cap \] A1. The
lock compatibility matrix now becomes as shown in Table 3.

**Semantics of Parameterized Access Modes**

At the very core of this approach to cooperative
transactions is the following observation: uncommitted data
contains unreliable information, i.e., it is associated with some
level of uncertainty. Thus, dealing with uncommitted data is
just a special case of the general problem of dealing with
uncertainty. Many papers have been published on this topic.
Dyreson (1993) provides a “bibliography of documents on
uncertainty and imprecision in information systems”
containing 352 references. Motro (1990ab) provides surveys.

According to (Motro 1990b) “most approaches to the
modelling of imprecision in databases are based on the theory
of fuzzy sets.” An alternative approach is to use probability
distributions, rather than the possibility distributions found in
fuzzy set theory (Barbara et al 1990, Feng et al 1991, Jia et al
1992). Both in fuzzy and probabilistic approaches, users will
have to associate with data attributes numbers in the interval
[0, 1] to denote degrees of membership or probabilities
respectively. Choosing numbers that faithfully reflect reality

<table>
<thead>
<tr>
<th></th>
<th>B</th>
<th>IR(A)</th>
<th>R(A)</th>
<th>U</th>
<th>IW(B)</th>
<th>R(A)IW(B)</th>
<th>W(B)</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>IR(A)</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>?</td>
<td></td>
</tr>
<tr>
<td>R(A)</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IW(B)</td>
<td>*</td>
<td>*</td>
<td>?</td>
<td>*</td>
<td>?</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R(A)IW(B)</td>
<td>*</td>
<td>?</td>
<td>?</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>W(B)</td>
<td>*</td>
<td>?</td>
<td>?</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3: Compatibility matrix for parameterized lock modes: * indicates compatibility, ? indicates compatibility iff B \[ \cap \] A, and ?? indicates compatibility iff B \[ \cap \] A both ways (i.e. from lock holder to
lock requester, and vice versa).
is clearly a non-trivial task, and Barbara et al (1990) acknowledge that this “could be a problem in practice because it may be difficult to know the exact probabilities.”

CCSR is therefore based on a different approach. Users should not be forced to quantify uncertainty, but should be allowed to classify it. That is, users can denote unreliable data as belonging to one or more of a predefined set of reliability categories. Each such category is represented by a reliability indicator, which should be defined in the metadatabase and may be used as an access mode parameter. The domain D of such parameters can be chosen so as to meet the needs of the applications at hand. Carefully note that the system should not know anything about the semantics of the members of D; only the applications should. Programmers must understand the semantics of the available parameters (as defined, e.g., by the database administrator), and use them accordingly. On the other hand, any need that applications may have for fine grained control over access modes may be accommodated by defining parameter values in D with the desired semantics, as long as the resulting run-time overhead and complexity of the environment remains acceptable. This is the main reason why CCSR is referred to as an application-oriented correctness criterion.

Relationship of CCSR to the Traditional Transaction Classes

CCSR differs from Conflict Serializability (CSR) in its definition of conflict. While all non-commutative operations are considered conflicting in CSR, read-write and write-read conflicts are made conditional in CCSR. The CSR notion of conflict is a special case of the CCSR notion of conflict, and it follows that the class of CSR histories must be a proper subset of the class of CCSR histories.

The following acronyms will be used below when referring to histories that are recoverable (RC), avoiding cascading aborts (ACA), strict (ST), and rigorous (RG) (Bernstein et al., 1987; Breitbart et al., 1991).

By means of suitable access mode parameters one transaction may read uncommitted data from another transaction and then commit before the other transaction. This is a violation of the RC condition, and shows that CCSR histories exist that are not RC. Such histories will of course not be ACA or ST either. Conversely, ST histories are not necessarily CCSR since ST does not prevent read-write conflicts. It follows that histories can also be ACA or RC without being CCSR.

The relationship between view serializability (VSR) and CCSR can be deduced by analysing how VSR and CCSR order conflicting operations. Observe that VSR and CCSR must order conflicting reads and writes the same way. More precisely, if a VSR history \(H_{VSR}\) and a CCSR history \(H_{CCSR}\) are both equivalent to a serial history \(H_s\), then all conflicting read and write operations will be ordered the same way in all three histories. \(H_{CCSR}\) must by definition order all conflicting operations the same way as in \(H_s\); \(H_{VSR}\) must order all conflicting read and write operations the same way as in \(H_s\), or else the read-from relationships will be different in the two histories. By transitivity, all conflicting read and write operations are ordered the same way in \(H_{VSR}\) and \(H_{CCSR}\). Thus, VSR and CSR histories can only differ in the way they order conflicting writes. Since there is no difference between CCSR and CSR when it comes to handling write-write conflicts, any history that is VSR and CCSR must necessarily also be CSR. That is, \(CSR \leftrightarrow VSR\). We also know that CSR \(\varphi VSR\) (Bernstein et al., 1987, 40) and CSR \(\varphi CCSR\), therefore CSR \(\perp VSR \leftrightarrow CCSR\). It follows that CSR = VSR \(\leftrightarrow CCSR\).

The relationships between RC, ACA, ST, RG, CSR, VSR and CCSR are illustrated in Figure 1.

CCSR and Recoverability-Related Properties

In traditional environments one wants to ensure both strictness (ST) and serializability of transaction histories, in the interest of recovery control and concurrency control respectively. This is achieved by means of rigorous (RG) histories; a proper subset of the intersection of ST and CSR. Similarly, one must consider how CCSR histories can also be made strict. If enforcing rigorosity is the only way of ensuring CCSR and ST, nothing seems to be gained. And given that the intersection of ST and CCSR is just a small subset of CCSR, and not much bigger than the intersection of ST and CSR anyway, one could easily think that the practical

Figure 1: Relationships between the traditional transaction history classes and the class of CCSR histories. All shown inclusions are proper. No rectangle is drawn for CSR; it is the intersection of CCSR and VSR. (Except for CCSR, this Venn diagram has been adopted from Bernstein et al. (1987) and Breitbart et al. (1991).)
usefulness of CCSR would be quite limited.

However, the key observation here is that the recovery-related properties of transaction histories are not independent of their concurrency-related properties; the notions of RC and ACA are really forced upon us by the correctness criterion for concurrency. If one transaction reads from another and also commits before that other one, the resulting history is non-recoverable. This is so because when a transaction aborts, the system must not only wipe out the effects of the aborting transaction, but also the effects of any transaction that read from the aborting transaction, i.e. those transactions must also be aborted (Bernstein et al 1987, 6; Hadzilacos 1983), the reason being that the resulting history would otherwise be non-serializable (non-VSR and thus also non-CSR). In other words, RC and ACA are actually functions of serializability, and will henceforth be denoted RC(SR) and ACA(SR) respectively. ST histories are motivated by the need for simple and efficient recovery algorithms, and as such have nothing to do with serializability. Analogous remarks apply to RG. However, since RG is a function of ST, which is a function of ACA, both ST and RG are indirectly functions of serializability, and will henceforth be denoted ST(SR) and RG(SR).

What must be done now is to define the recoverability properties for transaction histories modulo the new correctness criterion. Operations by a transaction $T_i$ is denoted $r_i$ for read, $w_i$ for write, $c_i$ for commit, $a_i$ for abort. The symbol $<_H$ is used to denote precedence in history $H$, e.g. $c_i <_H c_j$ means that $c_i$ precedes $c_j$ in $H$. The following definitions use the term conflicting operations in accordance with the above definition of that term. Recall that only operations by different transactions on the same data item can potentially be conflicting.

**Definition 2:** A history $H$ is recoverable modulo CCSR, denoted RC(CCSR), if for any conflicting operations $w_i <_H r_j$ we either have $c_i <_H c_j$ or $a_i <_H c_j$.

Informally, a history is RC(CCSR) if each transaction commits after the commitment of all transactions from which it performed conflicting reads.

**Definition 3:** A history $H$ avoids cascading aborts modulo CCSR, denoted ACA(CCSR), if for any conflicting operations $w_i <_H r_j$ we either have $a_i <_H r_j$ or $c_i <_H r_j$.

Informally, a history is ACA(CCSR) if no transaction performs any conflicting reads.

**Definition 4:** A history $H$ is strict modulo CCSR, denoted ST(CCSR), if it is ACA(CCSR) and for any conflicting operations $w_i <_H w_j$ we either have $a_i <_H w_j$ or $c_i <_H w_j$.

Informally, a history is ST(CCSR) if it is ACA(CCSR) and no data item written by one transaction can be overwritten by another transaction before the first transaction has aborted or committed.

**Definition 5:** A history is rigorous modulo CCSR, denoted RG(CCSR), if it is ST(CCSR) and for any conflicting operations $r_i <_H w_j$ we either have $a_i <_H w_j$ or $c_i <_H w_j$.

Informally, a history is RG(CCSR) if it is ST(CCSR) and no data item read by one transaction is ever overwritten in a conflicting mode by another transaction before the first transaction has aborted or committed.

The relationships between CCSR histories and the histories that fulfill the various recovery-related properties modulo CCSR, are obviously analogous to the relationships between the traditional classes of transaction histories. This is illustrated in Figure 2. The CSR, ST(SR), and RG(SR) classes are also shown. Detailed explanations for this figure are provided in Anfindsen (1997, 40-41).

**Enforcing both CCSR and Proper Recoverability**

Just as rigorous two phase locking (2PL) based on the traditional notion of conflict ensures RG(SR), rigorous 2PL based on the conditional notion of conflict (i.e. rigorous 2PL with parameterized locks) will ensure RG(CCSR), which is a (proper) subset of CCSR. This is so because rigorous parameterized 2PL will prevent all significant conflicts, which will cause CCSE(H) for the history $H$ in question to have no edges and therefore no cycles. Thus, by the if part of Theorem 1 $H$ will be CCSR.

**Prefix Commit-Closure**

Properties of transaction histories have little practical value unless they themselves have the property of being prefix commit-closed (PCC) (Bernstein et al., 1987, 36). It is shown below that the transaction properties defined in this paper are PCC. Commit and abort operations are collectively referred to as terminating operations. Operations by a transaction $T_i$ is denoted $r_i$ for read, $w_i$ for write, $c_i$ for commit, $a_i$ for abort, and...
Observation 1: Whenever \( a_i \neq H \), then \( T_i \) is without significance for prefix-commit closure since none of its actions will be in the committed projection of any prefix \( H' \) of \( H \). (Therefore, aborted transactions are not considered in the proofs below.)

Observation 2: For any non-terminating operation \( p_i \) by any committed transaction \( T_i \) in any history \( H \), \( p_i \prec_p c_j \).

Observation 3: If \( p_i \prec_p q_j \) and \( H' \) is any prefix of \( H \) and \( q_j \sim H' \), then \( p_i \prec_p q_j \).

Observation 4: If \( p_i \prec_p q_j \) and \( H' \) is any prefix of \( H \), then \( p_i \prec_p q_j \).

Theorem 2: RC(CCSR) is PCC, i.e., if \( H \) is RC(CCSR) then for any prefix \( H' \) of \( H \), \( C(H') \) is also RC(CCSR).

Proof: Since \( H \) is RC(CCSR) then for any conflicting operations \( w_i \prec_p r_j \) of committed transactions \( T_i \) and \( T_j \) we must have \( c_i \prec_p c_j \). For any prefix \( H' \) of \( H \), if \( c_j \prec_p H' \), then by observations 2 and 3 \( w_i \prec_p r_j \) and \( C(H') \) and \( w_i \prec_p r_j \) and \( C(H') \) if \( c_j \prec_p H' \), then \( T_j \) has no operations in \( C(H') \) and therefore cannot contribute to an ACA(CCSR) violation in \( C(H') \). It follows that RC(CCSR) is PCC.

Theorem 3: ACA(CCSR) is PCC, i.e., if \( H \) is ACA(CCSR) then for any prefix \( H' \) of \( H \), \( C(H') \) is also ACA(CCSR).

Proof: Since \( H \) is ACA(CCSR) then for any conflicting operations \( w_i \prec_p r_j \) of committed transactions \( T_i \) and \( T_j \) we must have \( c_i \prec_p c_j \). For any prefix \( H' \) of \( H \), if \( c_j \prec_p H' \), then by observations 2 and 3 \( w_i \prec_p r_j \) and \( C(H') \) and \( w_i \prec_p r_j \) and \( C(H') \) if \( c_j \prec_p H' \), then \( T_j \) has no operations in \( C(H') \) and therefore cannot contribute to an ACA(CCSR) violation in \( C(H') \). It follows that ACA(CCSR) is PCC.

Theorem 4: ST(CCSR) is PCC, i.e., if \( H \) is ST(CCSR) then for any prefix \( H' \) of \( H \), \( C(H') \) is also ST(CCSR).

Proof: Since \( H \) is ST(CCSR) then for any conflicting operations \( w_i \prec_p r_j \) we must have \( c_i \prec_p w_j \). For any prefix \( H' \) of \( H \), if \( c_j \prec_p H' \), then by observations 2 and 3 \( w_i \prec_p r_j \) and \( C(H') \) and \( w_i \prec_p r_j \) and \( C(H') \) if \( c_j \prec_p H' \), then \( T_j \) has no operations in \( C(H') \) and therefore cannot contribute to an ACA(CCSR) violation in \( C(H') \). It follows that ST(CCSR) is PCC.

Theorem 5: RG(CCSR) is PCC, i.e., if \( H \) is RG(CCSR) then for any prefix \( H' \) of \( H \), \( C(H') \) is also RG(CCSR).

Proof: Since \( H \) is RG(CCSR) then for any conflicting operations \( r_j \prec_p w_i \) we must have \( c_j \prec_p w_i \). For any prefix \( H' \) of \( H \), if \( c_j \prec_p H' \), then by observations 2 and 3 \( r_j \prec_p w_i \) and \( C(H') \) and \( r_j \prec_p w_i \) and \( C(H') \) if \( c_j \prec_p H' \), then \( T_j \) has no operations in \( C(H') \) and therefore cannot contribute to an RG(CCSR) violation in \( C(H') \). It follows that RG(CCSR) is PCC.

Theorem 6: CCSR is PCC, i.e., if \( H \) is CCSR then for any prefix \( H' \) of \( H \), \( C(H') \) is also CCSR.

Proof: Since \( H \) is CCSR then for any conflicting operations \( c_i \prec_p r_j \) of committed transactions \( T_i \) and \( T_j \) we must have \( c_i \prec_p r_j \) and \( C(H') \) and \( c_i \prec_p r_j \) and \( C(H') \) if \( c_i \prec_p H' \), then \( T_i \) has no operations in \( C(H') \) and therefore cannot contribute to an ST(CCSR) violation in \( C(H') \). It follows that ST(CCSR) is PCC. Therefore CCSR is PCC.

Example

Consider the designers Bob and Alice working on a new aeroplane. Let’s say Alice is working on the landing gear and Bob is working on the hydraulic system, with both sets of objects stored in a design database. Both Bob and Alice execute transactions that last for days or weeks. Given that the landing gear and electric system must be properly interfaced, Bob and Alice will both repeatedly need to look at the design objects of the other. This will cause cyclic write-read dependencies to be established between their transactions, which is incompatible with serializability.

Of course, one solution to this problem could be to allow dirty reads, but that would not be a good solution, one reason being that it provides no means of distinguishing between relatively stable design objects and those that are currently undergoing major changes. All sorts of inconsistencies can result from dirty reads, and the system will not provide the reader with any hints as to what they might be. In contrast, parameterized access modes make it possible for readers to filter away data that is unacceptably unreliable, and an application that retrieves uncommitted data receives information about the reliability of the data from the parameters that the writer has attached to it.

Assume that Alice is using a CAD system with transactional capabilities. Initially, the design is constantly changing. Alice probably has a lot of information about the new landing gear she is about to design; she might know that her company has signed a contract with a new supplier of tires, that a decision has been made to use a new hydraulic technology, etc. Whether she copies the landing gear of an existing aeroplane into her work space or she chooses to work from scratch, there will be a period of time when her design is unstable and inconsistent. During this period, it would make sense to use write parameters that indicate severe unreliability, and her CAD system could automatically make sure that this happens.

Sooner or later the design reaches a stage where it can be considered a complete draft; it is consistent and it meets overall design criteria that were given in advance (having to do, say, with weight and physical dimensions). This is when the need to cooperate with other designers comes into full play. She will need to look at the design of other parts of the aeroplane, such as the wings, the fuel tanks, the hydraulic system, the electric system, etc. Conversely, other designers will need to look at the landing gear design. A number of modifications to many parts of the aeroplane is to be expected, many of which will make other modifications necessary. For example, it might be necessary to alter the physical placement
of some hydraulic pipes, forcing Alice to modify the landing gear design accordingly.

Designing an entire aeroplane is an enormous task, and the work evolves through several stages. Presumably, there must be a company policy in place that specifies various levels of approval that designs can achieve. And e.g., the designer of the overall hydraulic system would be interested in knowing the level of approval achieved by designs that involve hydraulic components. This suggests that the various levels of approval that a given design goes through during its lifetime, should be reflected by the write parameters used at any given time. In other words, a CAD system should be able to represent levels of approval. For example, Alice may want to distinguish between the following classes of reliability for her design objects:

- incomplete draft (id)
- complete draft (cd)
- approved draft level 1 (ad1)
- approved draft level 2 (ad2)

Her data objects would initially be locked in W(id) mode, which would be changed to W(cd) mode as soon as the draft was complete. As her design gained the approval of group leader and later of project manager, say, W(ad1) and then W(ad2) would be used. Readers who are interested only in data objects that have at least reached the first level of approval, would use R(ad1, ad2) mode. As should be obvious from this example, when transactions last for a long time there is a need for locking algorithms that do not simply block whenever incompatible lock modes are encountered. A straightforward solution would be to interpret the use of a parameterized read mode as a request for data filtering, and simply skip encountered objects that are locked in incompatible write modes. Such a solution raises some questions having to do with the correctness of query results, a discussion of which is beyond the scope of this paper.

Assuming that the many engineers involved in designing an aeroplane are not always synchronized, i.e., that their designs may be at different levels of approval, one can easily conceive of managers and others wanting to produce reports that help them assess progress. Thus, it is probably a good idea to have a connection between reliability indicators and the query language supported by the system. This is discussed in Anfindsen (1997, 85-92). When CCSR is combined with nested databases a complete solution for collaborative transactions is provided, with customizable handling of read-write, write-read, and write-write conflicts.

Summary and Conclusions

A new correctness criterion, CCSR, based on a conditional notion of conflict has been defined. It makes it possible to modify the way read-write and write-read conflicts between transactions are handled. The relationship between CCSR and the traditional classes of transaction histories has been described, and recovery-related classes of transaction histories modulo CCSR has been defined. It was shown that CCSR can be enforced by rigorous, parameterized two-phase locking, and that CCSR as well as RC(CCSR), ACA(CCSR), ST(CCSR), and RG(CCSR) are all prefix-commit closed. Of the ACID properties only isolation is compromised by CCSR, which provides conditional isolation.

Uncommitted data is unreliable because it is subject to both rollback and further updates. Support for dirty reads has been available in DBMSs for a long time, but one reason why its usefulness is limited is that readers (in general) are given no information of the status or degree of reliability of the retrieved data. CCSR was motivated by the belief that a good framework for dealing with non-CSR transaction histories ought to provide readers with such information.

CCSR is a general-purpose correctness criterion and is not tailored to any specific application domain, and two of its main virtues are generality and simplicity. Some examples of potential usefulness can be found in the area of cooperative transactions, others can e.g. be found in waiting and blocking situations due to atomic commit protocols in distributed databases (Anfindsen,1995; 1997, 85-92). When CCSR is combined with nested databases a complete solution for collaborative transactions is provided, with customizable handling of read-write, write-read, and write-write conflicts.

References


Dr. Ole J. Anfindsen is a research scientist at Telenor R&D and an associate professor of computer science in the Software Engineering and Database Research Group, University of Oslo, Norway. He has been involved with database technology since 1982, and his research interests include database theory, transaction models, and object orientation. Anfindsen is the Technical Representative of Telenor to the Object Data Management Group (www.odmg.org). He can be reached at ole.anfindsen@fou.telenor.no.
Related Content

Quadtree-based Image Representation and Retrieval
www.irma-international.org/chapter/quadtree-based-image-representation-retrieval/29660/

A Hypertext Development Methodology
www.irma-international.org/article/hypertext-development-methodology/51118/

Fabric Database and Fuzzy Logic Models for Evaluating Fabric Performance
www.irma-international.org/chapter/fabric-database-fuzzy-logic-models/20367/

Extraction, Transformation, and Loading Processes
www.irma-international.org/chapter/extraction-transformation-loading-processes/7945/

Unified Modeling Language: A Complexity Analysis
Keng Siau and Qing Cao (2001). Journal of Database Management (pp. 26-34).
www.irma-international.org/article/unified-modeling-language/3259/