Real-time distributed concurrency control algorithms with mixed time constraints

Eindhoven University of Technology
Department of Mathematics and Computing Science

Real-Time Distributed Concurrency Control Algorithms with mixed time constraints

by

P.D.V. van der Stok

96/18

ISSN 0926-4515

Computing Science Report 96/18
Eindhoven, September 1996

Real-Time Distributed Concurrency Control Algorithms with mixed time constraints

P.D.V. van der Stok
Eindhoven University of Technology
Department of Computing Science
P.O. Box 513
5600 MB Eindhoven, Netherlands

August 7, 1996

Abstract

Two types of applications are considered: Hard Real-Time (HRT) and Soft Real-Time (SRT). HRT applications need to meet their dead-lines under all circumstances; deadlines of SRT applications may occasionally be missed. HRT and SRT applications coexist on the same set of processors and need to exchange data in a consistent manner.

Several concurrency control algorithms exist to assure the consistency of the results of transactions in distributed database systems. Three types of concurrency control are generally recognized: locking, timestamp ordering and optimistic concurrency control. They all impose an acyclic order on the individual actions of the transactions. The addition of versions to the data can diminish the execution time of read-only transactions. Imposing two different orderings, one for read-only transactions and one for all other transactions, the wait-time of read-only transactions is considerably reduced. The specified RO-Read allows the exchange of data between HRT and SRT applications without perturbations to the HRT applications.

1 Introduction

Real-Time application software reacts to stimuli from its environment. The state of the application is determined by the environment, while the state of the environment in its turn is determined by the state of the application. The application software (controlling process) often interacts quite heavily with a physical process (process under control).

The results of Real-Time applications not only need to be functionally correct, they must also be delivered in time. A result that is delivered too late is as bad as, or even worse than, a result that is not delivered at all. The timeliness apects of the Real-Time applications make Real-Time systems notoriously complex. In this paper, an application is divided into two parts: A Hard Real-Time (HRT) and a Soft Real-Time (SRT) part (application). Both applications have deadlines which have to be met. However, the missing of the deadline of a HRT application has catastrophic consequences for the process under control; the missing of a SRT application only constitutes a temporary degradation of the system performance. This division has been made to provide the system designer with a means to put some structure in its application to increase the number of designs that meet a given specification.

The object-oriented programming paradigm is followed for the development of the applications. An application consists of a program which is constructed from classes. A class defines a data-structure and the actions which can be executed on this structure. An instance of a class is called an object. Objects are distributed over a set of processors. Objects do not migrate between processors. The interleaving of actions to the same object may lead to inconsistent results. When during two concurrent accesses to the object, an item a with value A is read followed by the storage of $A + 1$ into a, a possible final result $A + 1$ differs from the intended result $A + 2$. To prevent such unintended outcome, objects can be declared atomic. This means that the results of the actions on an object are the same as if those actions were executed at one unique atomic moment of time. Inconsistent results caused by the interleaving of the actions are then excluded. The atomicity concept allows the system designer to concentrate on the functionality of his actions and not to bother about the intricate interactions between the different concurrent executions of these actions. The concept of atomic objects is motivated by the well-known concept of *Conversation Schemes* [AL81] used for the construction of reliable Real-Time systems. The duality of atomic objects and conversation schemes has been shown in [SMB85]. Concurrency Control Algorithms (CCA) impose an order on the concurrent actions on the objects with the purpose to meet the atomicity requirements of the objects. The research area of Real-Time (RT)-databases is lively one [Gra92, HSRT91]. The objective is to specify serialization criteria or CCA's to create transactions which meet their deadlines. The here considered objects are only stored in memory. Recovery of objects from stable storage is a different topic treated separately. This restriction is the major difference with RT-databases.

Transactions in RT-databases can be classified in three categories [Ram93]:

- Write-Only (WO)-transactions which read data from the environment and store them (in the database),
- Update transactions which read stored data and derive new data to be stored,
- Read-Only (RO)-transactions which read stored data and send them to the environment.

The addition of versions [Wei87] to objects increases the number of possible interleavings thus diminishing the probability that transactions need to be restarted and increasing the probability that they meet their deadlines. In this paper an algorithm based on versions is presented *which diminishes the time that (RO)-transactions wait for other transactions.* Consequently, the execution time of RO-transactions decreases.

Typically a large part of the WO- and RO-transactions, which interact directly with the environment need to be HRT-transactions. The other transactions can for a larger part be SRT-transactions. Data between HRT- and SRT-transactions is exchanged via objects. It is shown that the requirement of non-perturbation of HRT-transactions by SRT transaction considerably reduces the number of possible interleavings of SRT- and HRT-transactions. When the HRT-transactions are designed such that the order in which the results of HRT-transaction are delivered is more important than the serialization order imposed by the CCA used for the HRT-transactions, the concept of transfer serializability can be introduced. Transfer serializability is a correct criterium iff the transfer of object-modifications by a given HRT(SRT)-transaction to an equivalent set of objects readable by SRT(HRT)-transactions does not lead to unwanted system behaviour.

Within the actions of an object, invocations to actions of other objects can be specified. Such nested actions can also be declared atomic. However, the actions invoked by the atomic action should in their turn also be atomic. These actions are called nested atomic actions similar to the concept of nested transactions [Mos87].

In section 2 a historical view on database CCA's is presented, followed by a informal description of transaction properties in section 3. The first presented algorithm is an extension to Optimistic Concurrency Control with Time Intervals (OCC-TI) [LS93]. OCC-TI is a forward validation variation [HSRT91] on Optimistic Concurrency Control (OCC) [KR81]. The here presented Versioned OCC-TI (OCC-VTI) algorithm is an extension of OCC-TI with versions and modified for execution on a distributed computer platform. A specification and proof of OCC-VTI is presented in sections 5 and 6. The extension to HRT/SRT communication is made in section 7. The concept of nested transactions is discussed in section 8. The same additions as discussed for OCC-VTI are added to several CCA's developed for nested transactions, followed by their proof of correctness in section 9. The rest of the paper discusses implementation aspects of the different actions. A measure is calculated to decide the number of versions which are necessary to increase the performance of the RO-transactions.

2 Database historial

Actions on a database consist of a series of read or write actions on individual database elements. These actions are grouped in transactions, which can be executed in an interleaved fashion. An execution of a set of transactions is a serial execution iff for any pair of transactions T_i and T_j all actions of $T_i(T_j)$ are terminated before any action of $T_j(T_i)$ starts. Correct transactions executed in isolation, take the database from a given consistent state to a new consistent state. Consequently, starting from a consistent state, a serial execution of the transactions always leaves the database in a consistent state. In accordance, transactions executed in an interleaved fashion should only see values belonging to one and the same consistent database state and produce corresponding database values. The concept of serializabilty states that the result, delivered by a set of interleaved transactions, should be the same as a given serial execution of these transactions, called the *equivalent serial execution.* A lot of research has been done to specify the serializability criteria for databases. Simultaneously, research has been done on the algorithms which assure that the implementation of the transactions satisfies these serializability criteria [BHG87, Pap86].

Apart from serializability, another important aspect of databases is the permanency of the results even when failures occur such as processor or disk crashes [BHG87, GMA87]. The requirement that the database is always in a consistent state has as consequence that all actions of a transaction should be executed or none.

The concept of nested transactions was introduced to increase the parallellism inside a transaction and render recovery more efficient [Mos85, Mos87, BBG89]. The individual read and write actions no longer needed to be executed in a fixed serial order, but could be grouped and be executed in parallel. When a subtransaction of a transaction fails, only the subtransaction needs to be recovered.

The first concurrency control algorithms were based on locking. When two transactions simultaneously need to act on the same data-item and one of these actions is a write-action, the actions are said to *conflict.* When an action of one transaction conflicts with an action of another transaction, the transactions are said to be *conflicting.* When all conflicting actions are write-actions, the transactions are said to be *WW-conflicting.* When one of the conflicting actions is a read-action, the transactions are said to be *RW-conflicting.* The locking algorithm should assure that for any two conflicting transactions the conflicting actions of the one are executed before those of the other. The algorithms based on this principle are known to realize conflict serializability [Pap86]. The concept of two phase locking (2PL) assures this even in the case of processor or disk failures. 2PL states that a transaction first acquires locks in a growing phase and acquires none once it has released a lock. The main problem is the possibility of dead-lock. Notoriously difficult dead-lock detection and repair algorithms have been specified [Kna87].

Deadlock prevention assigns to each transaction a unique priority, usually defined by its start time. When a conflicting action is recognized by a transaction, lower priority transactions wait for higher priority transactions. When a lower priority transaction arrives first, only one of the two transactions is allowed to continue and the other aborts and restarts the execution of the transaction from scratch. As the priorities are totally ordered, no cycles can occur in the wait-for graph and deadlock is prevented. Deadlock prevention leads to transaction aborts which are not always necessary and to continuously aborted transactions (starvation).

The timestamp ordering (TSO)-approach completely abandons the lock concept. It is assumed that events in the distributed systems are totally ordered by logical clocks or synchronized clocks. Every transaction is assigned a timestamp that consists of the local time of the processor on which the transaction started suffixed with the processor identifier. Assuming that the processor-identifiers are totally ordered, also the timestamps are totally ordered. The CCA imposes the order prescribed by the timestamps on the equivalent serial order of the transactions. The algorithm aborts transactions or puts them in a wait state such that read actions only read values written by transactions with a lower timestamp and no values are written to data-items which have been read by transactions with higher timestamps than the timestamp of the writing transaction [Ree83]. Every data-item is typically equipped with a write-timestamp, that is the timestamp of the writing transaction, and a read-timestamp, that is the maximum of the timestamps of the transactions that have read this particular value of the item.

The drawback of this method is that frequently updated values make it difficult for slow transactions to read a value corresponding with its usually too low timestamp. The solution to this problem is the addition of versions, such that also slow Read-Only(RO)-transactions can proceed independent of the frequent updates of the values by other transactions. This approach is called Multi Version TSO (MVTSO) [Wei87]. Each data-item consists of a set of versions and each version consists of a read-timestamp and a write-timestamp as specified for the TSO algorithm.

For many databases, especially the large distributed ones, the number of conflicts will be relatively low. A more performant database system can then be created by assigning local copies of the database items to the transactions and let them execute their actions on these local copies. At the end of the transaction a serial order of the transactions is calculated and the validity of the values in the copies is verified against actions executed by other transactions on the same data-items. When an equivalent serial execution exists, the local copies are installed in the database. In case of invalid results, the copies are destroyed and the transaction is restarted. This approach is called Optimistic Concurrency Control (OCC) [KR81, Sch81]. At validation time, a timestamp is allocated to the transaction as explained for TSO and MVTSO. The algorithm assures that the order of an equivalent serial execution of the transactions corresponds with the order imposed by these timestamps. Two approaches exist: Backward Validation (BV) and Forward Validation (FV). In the BV variant, the validating transaction only commits when all commits in the past allow this, otherwise the validating transactions aborts. In the FV variant, the validating transaction aborts still executing transactions which will not be allowed to commit in the future. More recent OCC-FV algorithms determine on the basis of additional criteria (e.g. meeting of deadlines) whether the executing or the validating transaction should be aborted.

Most commercial database systems use the locking approach. The performance gain of the database system does not seem to outweigh the disadvantages of a new unchartered domain of concurrency control algorithms. However, Real-Time databases are gaining importance and the performance characteristics of the algorithms described above are investigated [Gra92, HSRT91]. The performance issue is relatively important in Real-Time systems. Typically, values will be updated with a relatively high frequency determined by the Real-Time environment. Relatively slow reads of the data-items follow a different time pattern. The introduction of versions is ideal to decouple the write frequencies from the read frequencies. However, in the inverse case, a frequent reading of the data at moments specified by the Real-Time environment can be seriously hampered by slow updates of these items. To solve this problem, an optimization of the RO-transactions supported by multiple versions is proposed in this paper. The performance improvement can be realized by specifying a different order for RO-transactions with respect to the other Writing(W)-transactions. RO-transactions will only read those versions of the data-items which have been validated by the writing transactions. This is enforced by allocating timestamps to RO-transactions equal to their start-time, and allocating timestamps to the W-transactions equal to their commit time plus an eventual constant.

3 Transactions

Concurrent actions on the atomic objects of a system meet the following two requirements:

- Concurrency atomicity: Actions behave as if they are executed in one unique point in time.
- Exception atomicity: All modifications of an action are visible to other actions or none.

The more frequently used term failure atomicity implies exception atomicity and moreover that all modifications to the database remain available in spite of processor and memory failures (permanency of results [BHG87, GMA87]). In this paper, permanency of results is not considered.

The concept of concurrency atomicity is equivalent with the isolation criterion used for transactions in databases. The serializability concept which characterizes transactions which conform to the isolation criterion applies equally well to the concurrency atomicity concept. Therefore, many results obtained by the research into RT-databases can be directly applied to atomic objects.

A database consists of a set of database items D. A database is *consistent* iff a database dependent predicate P holds. Atomic Read and Write actions can be executed on the individual items. Items are composed of versions which reflect the history of values atributed to these items. A transaction, T_i, is composed of a set of partially ordered actions. An execution of a set of transactions is called a *serial* execution iff for any two transactions T_i and T_j all actions of $T_i(T_j)$ are finished before any action of $T_j(T_i)$ starts.

When T_i executes a write action on item X it creates a version x_i notated with $W_i[x_i]$. When T_i reads an item, it reads the value of a version written by some transaction T_j, notated like: $R_i[x_j]$. Each element from I is read or written once by a given transaction. All read actions are executed before the write actions. This often justified simplification [Pap86] renders the algorithms and their accompanying proofs much simpler. An example of the execution of transaction T_i is $T_i = R_i[x_j]R_i[y_k]W_i[x_i]W_i[z_i]$. The values of j and k depend on the history of the data-items. A transaction T_i has a read set RS_i and a write set WS_i defined by: $RS_i = \{x_j \mid x_j$ is a version of $X \in D$ read by $T_i\}$ and $WS_i = \{x_i \mid x_i$ is a version of $X \in D$ written by $T_i\}$. A transaction only composed of read(write) actions is called a RO(WO)-transaction. A transaction with one or more write-actions is called a W-transaction. Two transactions which have versions x_i, x_j of the same data-item X as argument in one of their actions are called *related.* Two related transactions of which at least one contains a write-action to a common data-item are called *conflicting.*

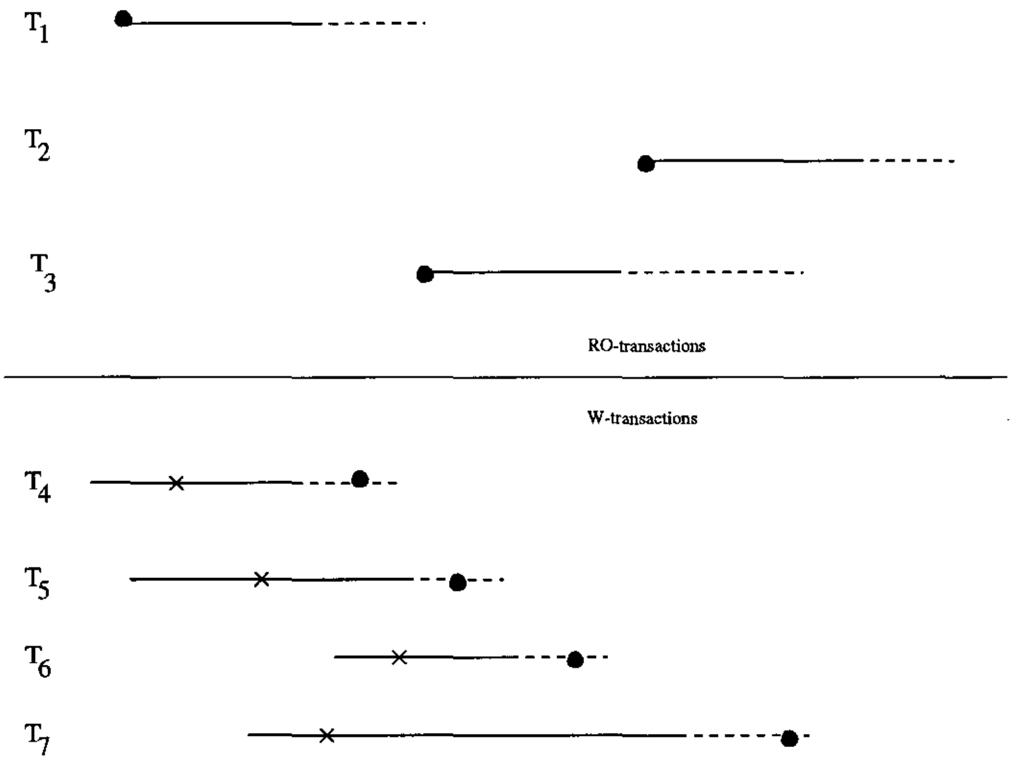

Figure 1: transaction times and ordering

In Fig. 1, transactions T_i, $0 < i \leq 7$, are shown which take a certain time as defined by the lines. A typical transaction is separated in a *read-write* phase denoted by the solid line and a *commit* or *validation* phase denoted with the broken lines. The transactions governed by the here presented algorithms behave with respect to RO-transactions as if they are executed in the timepoints denoted with large dots. W-transactions should behave with respect to W-transactions as if they are executed in the timepoints denoted with the crosses. The positions of the crosses and dots depend on the CCA. In current CCA's, the crosses and the dots coincide. With the presented RO variant of OCC-TI, the crosses are placed anywhere on the solid line. When a RO-transaction starts, the data for which no waits are needed are the data of transactions that are finished. Therefore, the dots of the W-transactions are preferably placed at the end of the broken lines and the dots of the RO-transactions are placed at the beginning of the solid lines. In fig. 1, the RO serial execution, determined by the dots, is $[T_1, T_4, T_3, T_5, T_6, T_2, T_7]$ and the W serial execution, determined by the crosses, is $[T_4, T_5, T_7, T_6]$. RO-transaction T_1 will read the results of none of the W-transactions $T_4 - T_7$ as the ordering prescribes that T_1 happens before $T_4 - T_7$. However, RO-transaction T_2 may read results of T_4, T_5 and T_6.

Problems may arise in the case of transactions T_6 and T_7. According to the crosses, T_6 happens before T_7, but according to the dots T_7 happens before T_6. Consequently, the values of the database-items as seen by the RO-transactions can be different from the ones seen by the W-transactions. For example assume that the transactions T_6 and T_7 act on three items [X,Y,Z] with consistency constraint $Z = X + Y$. Suppose that T_6 executes $Y := Y + 2; Z := Z + 2;$ and T_7 executes $X := X + 5; Z := Z + 5$. From the initial contents [2,3,5], the contents of [X,Y,Z] will either pass via [2,5,7] or via [7,3,10] to [7,5,12]. The two different orderings then impose that a given transaction can read the values [7,3,10], while another can read [2,5,7]. However, only one of the two possiblities is physically executed. Therefore, the set of predecessors of a RO-transaction is determined in the following way. All W-transactions with dots placed after the dots of the RO-transactions succeed the RO-transaction. All W-transactions which have crosses placed before the crosses of all transactions belonging to the successor set are members of the predecessor set. In the example the dot of T_7 is placed after the dot of T_2 and T_7 is part of the successor set of T_2. Transactions T_4 and T_5 are in the predecessor set of transaction T_2 because their dots are placed before the dot of T_2 and their crosses are placed before the cross of T_7. T_6 does not belong to the predecessor set because its cross is placed after the cross of T_7.

4 Informal description of OCC-VTI algorithms

The following rules apply for OCC-VTI. A transaction T_i precedes transactions T_j iff $T_i.C_f < T_j.C_f$. The following precedence relations are assured between a validating and X-writing T_v and the currently X-writing transaction T_w and X-reading transaction T_r: T_r precedes T_v precedes T_w [LS93]. No precedence relations are maintained between a X-reading T_v and T_r [LS93]. The precedence relation between a X-reading T_v and X-writing T_w is: T_v precedes T_w.

The algorithm of Fig. 2 determines which transactions read the values last written by a preceding transaction. A validity interval $[C_f, C_l]$ is attributed to the transactions and a validity interval $[C_b, C_e]$ is attributed to the versions of all items $X \in D$. The relation between the $[C_b, C_e]$ and the $[C_f, C_l]$ interval is the following. For a given version x_i written by T_i: $x_i.C_b = T_i.C_f$. The value of $x_i.C_e$ is given by: $x_i.C_e \geq \max\{T_j.C_f \mid T_j \textit{ has read } x_i\}$. For a given version x_i holds that: $x_i.C_b < x_i.C_e$. For two consecutively committed versions x_i and x_{i+1} holds: $x_{i+1}.C_b \geq x_i.C_e$.

The transaction is activated with the specification, RS and WS, of the actions it must execute and the data-items, I, on which they are executed. After the creation of the transaction identifier, $\mathcal{T}_s$, the validity interval is $[C_f, C_l]$ is initialised to $[0, \infty]$ and all read actions specified in RS are performed. The value read from each item X is stored in $A[X]$. Then the transaction can calculate results and store these into $A[X]$. The modified values are returned with the write actions specified in WS. At the end of the transaction Validate is executed in which the final C_f and C_l values are calculated and the end-time-stamp $\mathcal{T}_e$.

```
PROCEDURE W-Transaction(RS: read items, WS: write items, I: data-items)
 VAR
        A[data-items]: values
 BEGIN
        Create time-stamp $\mathcal{T}_s$ = time ∘ pid
        Create interval $[C_f, C_l] = [0, \infty]$
        $\forall X \in RS$ : A-read(X, $\mathcal{T}_s$, $A[X]$)
        execute calculations
        $\forall X \in WS$ : A-write(X, $\mathcal{T}_s$, $A[X]$)
        Create time-stamp $\mathcal{T}_e = 0$
        Validate(I, $\mathcal{T}_s, \mathcal{T}_e, C_f, C_l$)
        IF ($C_f < C_l$) THEN $\forall X \in I$: A-Commit( X, $\mathcal{T}_s, \mathcal{T}_e, C_f$)
        ELSE                  $\forall X \in I$: A-Abort( X, $\mathcal{T}_s$)
 END

PROCEDURE Validate(I: data-items, $\mathcal{T}_s, \mathcal{T}_e$ : Timestamp, $C_f, C_l$ : Counter)
 VAR
        $C_b, C_e$ : Counter
BEGIN
        $C_f := 0; C_l := \infty$
        $\forall X \in I$ DO
              A-Validate(X, $\mathcal{T}_s, t, C_b, C_e$)
              $C_f := \max(C_b, C_f); C_l := \min(C_l, C_e)$
              $\mathcal{T}_e = \max(\mathcal{T}_e, t)$
        OD
 END
```

Figure 2: W-transaction Algorithm

During the validation phase, a validating transaction, T_v, determines its own validity interval by taking the intersection of its validity interval $[C_f, C_l]$ with the validity intervals $[C_b, C_e]$ as returned by the A-Validate invocations.

When the Validate has succeeded, i.e. a non-empty interval has been found, T_v commits and the versions written by T_v are installed with the modifications to the C_b and C_e values of other versions. The time-stamp $\mathcal{T}_c$ attributed to each installed version for later use by the RO-transactions, is set equal to $\mathcal{T}_e$ that represents the maximum of the values returned by the A-Validate invocations.

The composing actions A-Validate, A-Read, A-Write, A-Abort and A-Commit are explained in more detail in section 5.

```
PROCEDURE RO-Transaction(I: data-items, t: time-stamp)
 VAR
        A[data-items]: values
 BEGIN
        Create time-stamp $\mathcal{T}_s$ = time ∘ PID
        IF ($\exists X \in I : (\exists x \in X : \mathcal{T}_c > \mathcal{T}_s) \vee x.S \neq Commit$) THEN
              $C_m := (\min\ X \in I : (\min\ x \in X, x.S \neq Commit \vee x.\mathcal{T}_c > \mathcal{T}_s : x.C_b))$
        ELSE $C_m := (\min\ X \in I : (\max\ x \in X : x.C_b))$
        $\forall X \in I$: RO-read(X, $C_m$, $A[X]$)
        execute calculations
END
```

Figure 3: RO-Transaction Algorithm

In Fig. 3, the algorithm for a RO-transaction is shown. At the start the identification time-stamp T_s is created. For all $X \in I$ with uncommitted versions or versions with a commit time-stamps larger than T_s, the minimum C_b value is determined. When no such versions exist the maximal C_b value is taken. This value is used to find the corresponding version as explained in section 3.

This distributed version of OCC-TI has a disadvantage with respect to the uniprocessor one discussed in [LS93]. In the uniprocessor case, validating transactions will succeed because the intervals of still executing transactions are adjusted possibly leading to their immediate abort. This is no longer possible because global distributed knowledge on all relevant $[C_b, C_e]$ intervals becomes available after the Validation phase.

5 Specification of OCC-VTI and RO-Read Actions

A given data-item X consists of a set of versions $X.V$. Each version x_i is a tuple :

$$< C_b, C_r, C_e, \mathcal{T}_c, \mathcal{T}_o, S, list, value > \qquad (1)$$

where (1) C_b is the begin counter of the validity interval, (2) C_r the maximum C_f of the transactions that read X, (3) C_e is the end counter of the validity interval, (4) $\mathcal{T}_c$ is the time-stamp of the commit action (5) $\mathcal{T}_o$ is the time-stamp that identifies the writer of the version, (6) S is the state of the version, (7) *list* is a list of identifiers of transactions that read this version and (8) *value* is the value of the item's version. The state S of a version is (1) *tentative*: the version is created but not yet usable by other transactions than the creating one, (2) *validating*: the transaction that created this version has finished all write and read actions but has not yet asserted their validity and (3) *commit*: the version can be used by transactions different from the creating one.

In Fig. 4, the specifications for the reading, writing, validating, and committing of one data-item are shown.

A-Read returns the value of the last committed version, v, and adds the identifier, $\mathcal{T}_i$, of the reading transaction to $v.list$.

A-Write adds a new version, v, to the set of versions, $X.V$. The specified value is stored in $v.value$ and $v.\mathcal{T}_o$ is set to the identifier of the writing transaction.

A-Validate is executed when no other transaction is validating the specified item, X. A-Validate calculates the authorised C_f, C_l value pairs for X. When the version, v, has been read, C_l is set equal to $v.C_e - 1$, otherwise C_l is set to ∞. When version v has been written, C_f is set equal to $b.C_r + 1$ where b is the last committed version of X. When no version has been written, a version v must have been read and C_f is set equal to $v.C_b + 1$ with v the read version. A local end timestamp is calculated and returned in t.

A-Commit sets the state of the written version, v, equal to Commit and $v.C_b$ equal to C_f. When a version v is written, $b.C_e$ is set to $v.C_f$ where b is the last committed version. When a version, v, is read, $v.C_r$ is set equal to the maximum of C_f and the former value of $v.C_r$.

A-Abort removes all versions, x, owned by the aborting transaction, $x.\mathcal{T}_o = \mathcal{T}_i$.

The specification of the RO-Read states that a committed version with the largest C_b value is selected that is smaller than the value, C, determined by the reading transaction (see Fig. 3).

```
ACTION A-Read( X: data-item, $\mathcal{T}_i$: Time-stamp, var a: value)
{PRE:    $v \in X.V \wedge v.S = commit \wedge v.list = L \wedge v.value = A$
         $\wedge(\forall x \in X.V, x.S = commit: \; x.C_b \leq v.C_b)$
POST:    $v.list = L \cup \{\mathcal{T}_i\} \wedge a = A$
}

ACTION A-Write( X: data-item, $\mathcal{T}_i$: Time-stamp, a: value)
{PRE:    $X.V = Q \wedge a = A$
POST:    $v$ = newly created version
         $\wedge v.S = tentative \wedge v.\mathcal{T}_o = \mathcal{T}_i \wedge v.value = A \wedge v.list = \emptyset \wedge X.V = Q \cup \{v\}$
}

ACTION A-Validate(X: data-item, $\mathcal{T}_i, t$: Time-stamp, var $C_f, C_l$: Counter)
{PRE:    $\forall x \in X.V : x.S \neq validating$
POST:    $\wedge(\forall x \in X.V, x.\mathcal{T}_o = \mathcal{T}_i : x.S = validating)$
         $\wedge((\exists v \in X.V : v.\mathcal{T}_o = \mathcal{T}_i) \Rightarrow C_f = (\max x \in X.V, x.S = commit : x.C_r + 1))$
         $\wedge((\forall v \in X.V : v.\mathcal{T}_o \neq \mathcal{T}_i) \Rightarrow C_f = (\max x \in X.V, \mathcal{T}_i \in X.list : x.C_b + 1))$
         $\wedge((\exists v \in X.V : \mathcal{T}_i \in v.list) \Rightarrow C_l = v.C_e - 1)$
         $\wedge((\forall v \in X.V : \mathcal{T}_i \notin v.list) \Rightarrow C_l = \infty)$
         $t = time \circ pid$
}

ACTION A-Commit( X: data-item, $\mathcal{T}_i, \mathcal{T}_e$: Time-stamp, $C_f$: Counter)
{PRE:    $(\forall v \in X.V : v.C_r = C_v)$
         $\wedge b \in X.V \wedge b.S = commit \wedge (\forall x \in X.V, x.S = commit : x.C_b \leq b.C_b)$
POST:    $((\exists x \in X.V : x.\mathcal{T}_o = \mathcal{T}_i) \Rightarrow b.C_e = C_f)$
         $\wedge(\forall v \in X.V, v.\mathcal{T}_o = \mathcal{T}_i : v.S = commit \wedge v.C_e = \infty \wedge v.C_b = C_f \wedge v.C_r = C_f \wedge v.\mathcal{T}_c = \mathcal{T}_e)$
         $\wedge(\forall v \in X.V, \mathcal{T}_i \in v.list : v.C_r = \max(C_v, C_f))$
}

ACTION A-Abort( X: data-item, $\mathcal{T}_i$: Time-stamp)
{PRE:    $X.V = Q \wedge C = \{v \mid v \in X.V \wedge \mathcal{T}_i = v.\mathcal{T}_o)\}$
POST:    $X.V = Q\text{-}C$
}

ACTION RO-Read( X: data-item, C: Counter, var a: value)
{PRE:    $v \in X.V \wedge v.C_b < C \wedge v.S = commit \wedge v.value = A$
         $\wedge(\forall x \in X.V : (x.C_b \leq v.C_b \vee x.C_b > C))$
POST:    $a = A$
}
```

Figure 4: OCC-VTI read, write, validate, commit and abort specifications

6 Proof of correctness

A serial history $h' = < T_0, T_1, T_2, ..., T_n, T_f >$ of transactions is represented by an ordered list of transactions $T_i, 0 < i \leq n$, preceded by a dummy transaction T_0 = W[D] and terminated by a dummy transaction T_f = R[D]. A transaction set H consists of the transactions $T_i, 1 \leq i \leq n$, joint with the set $\{T_0, T_f\}$. The set H is composed of a set of RO-transactions H^{ro} and a set of W-transactions H^w, with $H = H^w \cup H^{ro}$ and $H^w \cap H^{ro} = \emptyset$. A write by a transaction T_i to a version x_i, that is not read by any transaction T_j is called a useless write, denoted by T_{ix}. The set H^u is defined by { T_{ix} | none of the transactions $T_j \in H$ reads the value x_i written by T_i}. The *read-from* relation rf(h) in history, h, is defined by: { (T_i, x_i, T_j) | T_j reads the value x_i written by T_i }.

It must be verified that a history h of read and write actions, as ordered by the algorithms and by the partial order prescribed by the transactions, is equivalent with a serial history

h' of these same transactions. This is proven by making a directed graph G with vertices $V = H \cup H^u$ and edges E, which represent the causal order of the read and write actions on individual items of D. For a given edge $e \in E$, $s(e)$ is the source vertex and $d(e)$ is the destination vertex. In [Vid85] it is shown that the graph of view serializable histories has certain characteristics. Edges are added to the graph G to demonstrate that the graphs describing the histories enforced by the CCA's have these characteristics.

Before the correctness proof, a few hypotheses are made about the implementation of the actions described in section 4. Motivated by the distribution of the data-items over the processors, it is assumed that a monotonously increasing clock is associated with each data-item. These clocks represent the local times of the data-items. A generally accepted hypothesis on the specified actions needs to be stated.

Hypothesis 1 *The ACTIONs specified in section 4 are executed atomically.*

It is important that A-Validate and A-Commit constitute one concurrency atomic action. The following example will show that when this hypothesis is not met, inconsistent states can be reached. Suppose two transactions T_1 and T_2 access two items $\{X,Y\}$. The transactions are defined by $T_1 = R_1[Y]\ W_1[X]$ and $T_2 = R_2[X]\ W_2[Y]$ and they execute according to the unserializable schedule $h = R_1[x_0]\ R_2[y_0]\ W_1[x_1]\ W_2[y_2]$. Both read the initial values x_0 and y_0 and want to install the values x_1 and y_2. Both transactions first validate. The validation result is TRUE for both transactions having read the last installed values x_0 and y_0. The ensuing commit of both transactions installs x_1 and y_2, which is an unwanted result as in a serial schedule either T_1 reads y_0 and T_2 reads x_1 or T_2 reads x_0 and T_1 reads y_2. Therefore, a hypothesis is made on the atomicity of the validate and commit actions.

Hypothesis 2 *For all $X \in D$ and for any transaction T, the action pair A-Validate(X, T_i, t, C_f, C_l) and A-Commit(X, T_i, T_e, C_f) constitutes one atomic action.*

A hypothesis is needed about the commit timestamps of X-writing transactions with respect to X-reading RO-transactions. For a given RO-transaction, T_{ro} and a W-transaction, T_w, which conflict in X, the following hypothesis is made:

Hypothesis 3 *If T_{ro} reads X at local time t and T_w commits at local time $t' > t$ then $T_{ro}.T_s < T_w.T_e$.*

The *transaction-graph* G is constructed as described in [Vid85]. The edge set E of G has the following characteristics:

- An edge labeled X from T_i to T_j for each $(T_i, x_i, T_j) \in rf(h)$, called *useful X-edge.*
- An edge labeled X from T_i to T_{ix} for each $T_{ix} \in H^u$, called *useless X-edge.*
- An unlabeled edge for each vertex $T_{ix} \in H^u$ to T_f.
- An unlabeled edge from each RO-transaction other than T_f to T_f.
- An unlabeled edge from T_0 to each WO-transaction.

Definition 1 (TP(h)) *An acyclic Transaction Precedence graph TP(h) of history h has the following characteristics:*

- *The vertex set of TP(h) is the same as the vertex set G*
- *The edge set of TP(h) includes the edge set of G joint with a (possibly empty) set of unlabelled edges.*
- *For any two edges e, e', where e is a useful X-edge and e' another (useful or useless) X-edge with $s(e) \neq s(e')$, there exists a directed path in TP(h) that contains them both.*

Theorem 1 *[Vid85] A history h is serializable iff there exists an acyclic TP(h).*

The graph G is constructed as described above. Every transaction $T_i \in H$ has a start-timestamp $\mathcal{T}_s$ and an end timestamp $\mathcal{T}_e$ such that $T_i.\mathcal{T}_s < T_i.\mathcal{T}_e$. $\mathcal{T}_s$ and $\mathcal{T}_e$ of transactions T_0 and T_f are defined such that:

$$\forall T_i \in H \setminus \{\mathcal{T}_o, T_f\} : T_0.\mathcal{T}_s < T_i.\mathcal{T}_s < T_f.\mathcal{T}_s \wedge T_0.\mathcal{T}_e < T_i.\mathcal{T}_e < T_f.\mathcal{T}_e \qquad (2)$$

.

RO-transactions read their data from one or more W-transactions or T_0. For a given RO-transaction, T_{ro}, a set of versions exists associated with the read set RS_{ro}. The maximum $\mathcal{M}_{r0}$ is defined as the minimum of all C_b-values of uncommited versions and the versions for which their commit-timestamp $\mathcal{T}_c > T_{ro}.\mathcal{T}_s$:

$$\mathcal{M}_{ro} = \min X \in RS_{ro} : (\min x \in X, (x.\mathcal{T}_c > T_{ro}.\mathcal{T}_s \vee x.S \neq \mathit{commit}) : x.C_b) \qquad (3)$$

Otherwise: $\mathcal{M}_{ro} = (\min X \in I : (\max x \in X : x.C_b))$. An X-order, $\overset{X}{\prec}$, can be defined for the transactions.

Definition 2 (X-order) *For any two transactions T_i and T_j, $T_i \overset{X}{\prec} T_j$ iff*

- *T_i and T_j are conflicting W-transactions with $X \in RS_i \cap WS_j \vee X \in WS_i \cap RS_j$ and $T_i.C_f < T_j.C_f$.*
- *$(i = 0 \wedge X \in RS_j)$ or $(j = f \wedge X \in WS_i)$.*
- *T_i is a W-transaction and T_j is a RO-transaction and $X \in WS_i \cap RS_j$ and $T_i.C_f \leq \mathcal{M}_j$.*
- *T_j is a W-transaction and T_i is a RO-transaction and $X \in WS_j \cap RS_i$ and $\mathcal{M}_i < T_j.C_f$.*

The consistency of these relations is proven in the following lemma:

Lemma 1 *The C_f values of any two conflicting transactions T_i and T_j differ.*

Proof. Suppose the transactions conflict in X. Suppose that both transactions are writing transactions. Assume without loss of generality that T_i has committed version, x_i after T_j validates and commits version x_j. In that case $x_j.T_o = T_j$ and the post condition of A-Validate states:

$$C_f = (\max x \in X.V, x.S = \mathit{commit} : x.C_r) + 1 \qquad (4)$$

together with the calculation of the maximum C_f value in the Procedure Validate, the calulation of C_r in the pre and post-conditions of A-Commit:

$$\begin{array}{ll} \text{PRE:} & (\forall v \in X.V : v.C_r = C_v) \\ \text{POST:} & (\forall v \in X.V, T_i \in v.\mathit{list} : v.C_r = \max(C_v, C_f)) \end{array} \qquad (5)$$

and the postcondition in A-Commit:

$$(\forall v \in X.V, v.T_o = T_i : v.S = Commit \wedge v.C_e = C_f \wedge v.C_b = C_f) \tag{6}$$

assure that:

$$T_i.C_f = x_i.C_b \leq x_i.C_r < x_j.C_b = T_j.C_f \tag{7}$$

This holds for all $X \in D$. Consequently, the C_f values of T_i and T_j differ when they write to the same item.

Again without loss of generality assume that T_i writes version x_i of X and T_j reads X. When T_j reads x_i: $x_i.T_o \neq T_j$. The post-condition of A-Validate:

$$(\forall v \in X.V : v.T_o \neq T_j \Rightarrow C_f = (\max x \in X.V, T_j \in X.list : x.C_b + 1)) \tag{8}$$

assures that $T_j.C_f > x_i.C_b = T_i.C_f$.

T_j can read a version x_p, with $x_p \neq x_i$. According to the post-condition of A-Read: $T_j \in x_p.list$. Suppose $x_p.C_b > x_i.C_b$. Similar to the reading of x_i: $T_j.C_f > x_p.C_b > x_i.C_b = T_i.C_f$. Suppose that $x_p.C_b < x_i.C_b$. In this case T_j reads x_p before T_i commits. If T_i had committed before T_j's reading of X, T_j would have read x_i or another after x_i committed version. Two possibilities exist: T_i commits after T_j or before. When T_i invokes A-Commit before T_j, the pre- and post-conditions of A-Commit:

$$\begin{array}{ll} \text{PRE:} & b \in X.V \wedge b.S = commit \wedge (\forall x \in X.V, x.S = commit : x.C_b \leq b.C_b) \\ \text{POST:} & (\exists x \in X.V : x.T_o = T_i) \Rightarrow b.C_e = C_f \end{array} \tag{9}$$

assure that $x_p.C_e \leq T_i.C_f$. The post condition of A-Validate assures that $T_j.C_f \leq T_j.C_l < x_p.C_e \leq T_i.C_f$.

When T_j invokes A-Commit before T_i, the post- and pre-conditions of A-Commit:

$$\begin{array}{ll} \text{PRE:} & (\forall v \in X.V : v.C_r = C_v) \\ \text{POST:} & (\forall v \in X.V, T_j \in v.list : v.C_r = \max(C_v, C_f)) \end{array} \tag{10}$$

assures that for the entry $T_j \in x_p.list$: $x_p.C_r \geq T_j.C_f$. The post condition of A-Validate assures that $T_j.C_f \leq x_p.C_r < T_i.C_f$.

Consequently, the C_f values of any two conflicting T_i and T_j differ. □

Lemma 2 *For two related transactions T_i and T_j, with $\{X, Y\} \subseteq (RS_i \cap WS_j) \cup (RS_j \cap WS_i)$: $T_i \overset{X}{\prec} T_j \Leftrightarrow T_i \overset{Y}{\prec} T_j$*

Proof. Assume both transactions to be conflicting W-transactions. According to lemma 1 the C_f-values differ and according to the order definition 2 the transactions are ordered. The C_f values are independent of the item in which they conflict. Consequently, the lemma is true for W-transactions.

When both transactions are RO-transactions, no order is defined and the lemma is trivially true.

Consider that $T_j \in H^{ro}$ and $T_i \in H^w$. No distinction exists between X and Y. Consequently, it is sufficient to prove that: $T_i \overset{X}{\prec} T_j \Rightarrow T_i \overset{Y}{\prec} T_j$. In the case $T_i \overset{X}{\prec} T_j$, the third bullet in definition 2 applies. Because $\{X, Y\} \subseteq WS_i \cap RS_j$, X can be replaced by Y. This

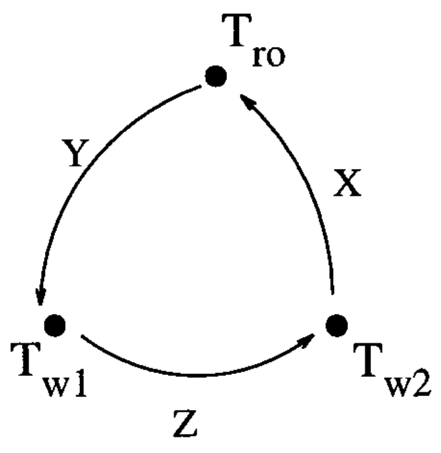

Figure 5: impossible cycle in transaction order

is exactly the definition for $T_i \overset{Y}{\prec} T_j$. The case that $T_j \in H^w$ and $T_i \in H^{ro}$ is treated in the same way. □

Lemma 3 *The order of definition 2 is acyclic.*

Proof. According to lemma 1 and the ordering of the numbers assigned to the C_f values, the sets of conflicting W-transactions are ordered without cycles. No order is defined between RO-transactions, but the ordering of RO-transactions with respect to W-transactions is different from the ordering between W-transactions. Consequently, cycles can occur when a RO-transaction T_{ro} is ordered between two W-transactions, T_{w1} and T_{w2}, such that $T_{w1} \overset{Z}{\prec} T_{w2} \wedge T_{w2} \overset{X}{\prec} T_{ro} \overset{Y}{\prec} T_{w1}$, as shown in fig 5. From definition 2, the following relations can be derived:

$$T_{w2} \overset{X}{\prec} T_{ro} \Rightarrow T_{w2}.C_f < \mathcal{M}_{ro} \tag{11}$$

$$T_{ro} \overset{Y}{\prec} T_{w1} \Rightarrow \mathcal{M}_{ro} \leq T_{w1}.C_f \tag{12}$$

Combining both relations leads to: $T_{w1}.C_f > T_{w2}.C_f$. Because $(Z \in RS_{w1} \cap WS_{w2}) \vee (Z \in WS_{w1} \cap RS_{w2})$, it follows that $T_{w2} \overset{Z}{\prec} T_{w1}$ and the cycle is impossible. □

The following abbreviation is used:

$$T_i \prec T_j \equiv \exists X \in D : T_i \overset{X}{\prec} T_j \tag{13}$$

In the relation $T_i \prec T_j$ both T_i and T_j can be freely exchanged with $s(e)$, $s(e')$, $d(e)$ and $d(e')$.

A transaction T_j is the *immediately related* successor to a transaction T_i iff $T_i \overset{X}{\prec} T_j$ and there is no other transaction T_k such that $T_i \overset{X}{\prec} T_k \overset{X}{\prec} T_j$. The graph G' is constructed from

G by adding edges between related transactions. From each vertex T_i to its immediately related successors T_{i+k} and to T_{ix} and from T_{ix} to its immediately related successor T_{i+k}. It will be proven that the graph G' represents a serializable history for the transactions which constitute h. In Fig. 6, an example of G' is drawn. A history h is assumed for the transactions $T_0, T_{10} - T_{13}, T_f$, where each transaction consists of the following read and write actions:

$$\begin{aligned} T_{10} &= R_{10}[x_0]\ R_{10}[y_0]\ W_{10}[z_{10}] \\ T_{11} &= R_{11}[x_0]\ R_{11}[z_{10}]\ W_{11}[z_{11}] \\ T_{12} &= R_{12}[x_0]\ W_{12}[y_{12}] \\ T_{13} &= R_{13}[y_{12}]\ W_{13}[z_{13}] \end{aligned} \tag{14}$$

The following history, h, of actions is associated with the 4 transactions:

$$R_{10}[x_0]\ R_{12}[x_0]\ R_{10}[y_0]\ R_{11}[x_0]\ W_{10}[z_{10}]\ R_{11}[z_{10}]\ W_{12}[y_{12}]\ R_{13}[y_{12}]\ W_{11}[z_{11}]\ W_{13}[z_{13}] \tag{15}$$

The graph G is represented in fig. 6 by the closed arrows. Remark that $W_{11}[z_{11}]$ is a useless write. The graph G' is constructed by adding the dashed arrows to G. An arrow is added from T_{10} to T_{12} because they are immediate successors via Y and from T_{11}, T_{11z} to T_{13} because they are immediate successors via Z.

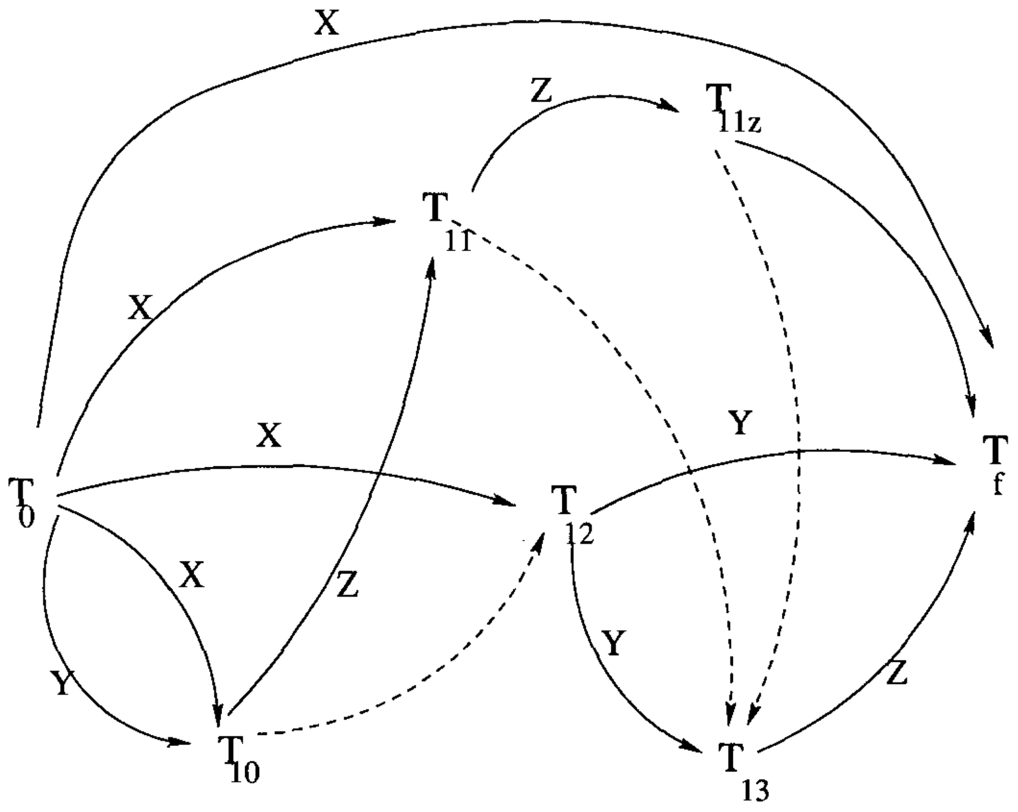

Figure 6: Example of graph G'

Lemma 4 *For any labelled edge e from G, with $s(e) = T_i$ and $d(e) = T_j$, T_i and T_j related via X, it holds that $T_i \overset{X}{\prec} T_j$.*

Proof. A RO-transaction T_{ro} is the destination of a given edge e, or is the source of an edge with destination T_f. In the first case, the pre-condition in RO-Read and the assignment to C in RO-Transaction:

$$v.C_b < C \wedge v.S = commit \wedge C = \mathcal{M}_{ro} \tag{16}$$

assures that $s(e).C_f < \mathcal{M}_{ro}$ and $s(e)$ precedes $d(e)$ in the defined order.

In the second case, T_f follows all other transactions per definition. Consequently, the lemma is true for all edges involving a RO-transaction. The same reasoning is valid for any edge involving $T_{ix} \in H^u$.

Edges which only involve W-transactions are now considered. The version $x_{s(e)}$ written and committed by $s(e)$ is read by $d(e)$. The post-condition of A-Read assures that an entry $d(e)$ is added to $x_{s(e)}.list$. The POST condition of A-Validate:

$$(\forall v \in X.V : v.T_o \neq T_i) \Rightarrow C_f = (\max x \in X.V, T_i \in X.list : x.C_b + 1) \tag{17}$$

returns a C_f values that is larger than $x_{s(e)}.C_b = s(e).C_f$. The maximum of the returned C_f values is calculated by the Validate Procedure. It can be infered that: $d(e).C_f > x_{s(e)}.C_b = s(e).C_f$.

From the order-definition 2 it follows that the source of a labelled edge is ordered before its destination. □

Theorem 2 (View serializability) *The graph G' constructed from the ACTION specifications of section 4 is an acyclic TP(h).*

Proof. By construction, the vertex set of G is equal to the vertex set of G' and the edge set of G' is equal to the edge set of G with the addition of some unlabelled edges. An acyclic order is defined on all transactions T_i and for all $X \in D$ there is a directed path in G' that includes all X-edges. For two X-edges e and e' with different sources of which at least one is useful, their destinations are different, $d(e) \neq d(e')$, because a transaction reads a given value only from one transaction at the time. From lemma 4 it is known that $d(e) \succ s(e)$ and $d(e') \succ s(e')$.

Suppose that one of them, e' is useless. The case $s(e) \prec s(e')$ is proven in the same way as the case that both edges are useful. When $s(e') \prec s(e)$, $s(e)$ is either the direct successor of $s(e')$ or ordered after the direct successor of $s(e')$. The destination of the useless write, $d(e')$, is defined to be ordered before the direct successor of the source: $d(e') \prec s(e)$.

Suppose both e and e' are useful edges. In that case $s(e)$ and $s(e')$ are both W-transactions because no labelled edges exist from a RO-transaction or a from a vertex $T_{ix} \in H^u$. Without loss of generality it is assumed that $s(e) \prec s(e')$. When $d(e) = s(e')$, the theorem is trivially true by taking the X-path from T_0 to $s(e)$ and via $d(e) = s(e')$ to $d(e')$ and finally to T_f. When $d(e) \neq s(e')$, the case that $d(e)$ is a W-transaction is treated first, followed by the case that $d(e)$ is a RO-transaction.

Assume $d(e)$ is a W-transaction. The proof is similar to the proof of lemma 1 when $d(e)$ reads a version x_p with $x_p.C_b < x_i.C_f$ with x_i written by $T_i = s(e')$. It was demonstrated that $s(e')$ commits after the reading of X by $d(e)$. Two cases were considered: (1) $d(e)$ commits after $s(e')$ and (2) $d(e)$ commits before $s(e')$. In both cases it was demonstrated that $d(e).C_f < s(e').C_f$.

Consider the case that $d(e)$ is a RO-transaction. When $s(e')$ is committed before $d(e)$ reads, then according to the PRE-condition of RO-Read:

$$\forall x \in X.V : (x.C_b \leq v.C_b \vee x.C_b > C) \tag{18}$$

and the selection of $C = \mathcal{M}_{d(e)}$:

$$s(e').C_f = x_{s(e')}.C_b > \mathcal{M}_{d(e)} \tag{19}$$

When $s(e')$ commits after the reading by $d(e)$, $s(e')$ also commits after $s(e)$ and according to the post-condition of A-Validate $s(e').C_f = x_{s(e')}.C_b > x_{s(e)}.C_b = s(e).C_f$. According to hypothesis 3: $s(e').\mathcal{T}_e > d(e).\mathcal{T}_s$, which means that the value of $\mathcal{M}_{d(e)}$ before and after the commit of $s(e')$ is the same and $s(e').C_f > \mathcal{M}_{d(e)}$. According to the order of definition 2: $d(e) \prec s(e')$ in both cases.

From the ordering and the additional unlabelled edges added to G', an acyclic path can be constructed started at T_0 from $s(e)$ to $d(e)$, from $d(e)$ to $s(e')$ and from $s(e')$ to $d(e')$ to end at T_f. □

7 HRT/SRT communication

A Real-Time environment puts constraints on the moments actions have to be executed. Starting from the requirement that no actions are better than too late actions, each transaction will have a deadline attribute. Transactions that do not finish before their deadlines loose their usefulness and may be aborted.

A careful scheduling of the HRT transactions should prevent that HRT-transactions interfere with each other in such a way that some deadlines are not met, or that preconditions cannot be met. For the purpose of this paper, it is assumed that such a scheduling of HRT transactions is actually possible albeit imperfectly from an efficiency point of view. The separation between HRT- and SRT- transactions, as introduced in section 1, necessitates the formulation of conditions under which objects can be accessed by both HRT- and SRT-transactions.

During the execution of the HRT schedule the meeting of the deadlines of the HRT-transactions is guaranteed as long as the schedule is not perturbed by the execution of the SRT-transactions. Consider two transactions T_i and T_j with T_i of a different type (HRT or SRT) from T_j. In the histories not only the Read and Write actions but also the Commit actions are included. $C[X]$ stands for the commit of data-item $X \in D$ and $C[Q]$ for the commit of the set of data-items $Q \subseteq D$. The notation $t(A_i[X])$ stands for the time on the local clock of item X at the moment of excution of action (A = R, W or C) by transaction T_i.

Lemma 5 *The HRT-transaction, T_i (T_j), is not perturbed by the SRT-transaction, T_j (T_i), if transactions are ordered such that: $t(R_j[X]) < t(C_i[X]) \Rightarrow T_j \prec T_i$.*

Proof. Consider the case that T_i is a HRT-transaction and T_j a SRT transaction. At the commit of the HRT transaction T_i, T_i must be aborted if T_i has written a version x_i to be read by an already committed T_j. This is the case when $t(R_j[X]) < t(C_i[X]) \wedge T_i \prec T_j$. The abort of a HRT transaction should be avoided and therefore $t(R_j[X]) < t(C_i[X])$ implies that $T_j \prec T_i$.

Consider the case that T_j is a HRT transaction and T_i a SRT-transaction. Consider two transactions: $T_i = ...W_i[x_i]\,C_i[X]$ and $T_j = ...R_j[X]\,C_j[X]$. A possible history, with $T_i \prec T_j \wedge t(R_j[X]) < t(C_i[X])$, is given by: $h = ...W_i[x_i]\,R_j[x_i]\,C_j[X]\,C_i[X]$. In this case T_j should wait for the commit of T_i before it can read the value x_i written by T_i. As HRT-transactions should not wait for SRT-transactions (with unknown durations), T_j should have read an earlier value of X and $T_j \prec T_i$ should hold. □

For the following take $H^h(H^s)$ as the set of HRT(SRT)-transactions, Q is the set of items accessed by transactions from both H^h and H^s. Take $T_h \in H^h$, $T_s \in H^s$ and $p, q \in Q$. From the above lemma the following theorem can be concluded.

Theorem 3 *For all T_h, T_s, q and p, with $T_h = ...R_h[q]....C_h[q,p]$ and $T_s = ...R_s[p]....C_s[q,p]$, no interleaving is possible between the Read action and the Commit action.*

Proof. Assuming the theorem is not valid, try to construct a valid interleaved history. Suppose the $R_s[q]$ is interleaved between $R_h[q]$ and $C_h[q,p]$. The action $C_s[q,p]$ can be scheduled before or after $C_h[p,q]$. Independent of the position of $C_s[q,p]$, an unwanted situation is created according to lemma 5 :

$$t(R_h[q]) < t(C_s[p,q]) \Rightarrow T_H \prec T_S$$
$$t(R_s[p]) < t(C_h[p,q]) \Rightarrow T_S \prec T_H$$

This constitutes a cycle and leads to a non view-serializable history. The interleaving of $R_h[q]$ between $R_s[p]$ and $C_s[q,p]$ leads to the same contradiction. Consequently, the only valid history is one without the discussed interleaving. □

A simple and valid way to solve this problem is to execute HRT- and SRT transactions in a perfect serial order. SRT-transactions which are preempted by HRT transactions must be aborted. The scheduling of the SRT transactions such that they never are executed concurrently with conflicting HRT-transactions restricts the possible schedules enormously and leads to many SRT-transaction aborts when conflicting HRT transactions are started during the SRT-transaction execution. Another option is to relax the view-serializability requirements. When HRT-transactions deliver sets of values such that not the serial order chosen by the CCA is important for the SRT-transactions but the order in which these sets are committed, another serializability criterium can be formulated. The same is true for values delivered by SRT transactions to HRT transactions. The concept of *transfer serializability* is introduced:

Definition 3 (Transfer-serializability) *A history of HRT- and SRT-transactions is transfer serializable iff in an equivalent view-serializable history the values written by a given HRT-transaction are transferred by a single transaction to reading SRT-transactions and vice-versa.*

A TR-Read action must be defined which obeys lemma 5. The RO-Read action does not influence the other W-transactions. Consequently, a modified version of the RO-Read action, shown in Fig. 7 as TR-Read, can be used by the SRT-transactions when the HW-objects are read. However, when a HRT-transaction invokes the TR-Read action on a data-item containing uncommitted versions for which the commit is already started, the transaction is obliged to wait until the commit is finished. The use of TR-Read by HRT-transactions to read SW-objects prevents such intolerable waits when the time to execute a commit of a SRT-transaction is bounded. With bounded commit execution times the $\mathcal{T}_c$ values can be chosen such that no waiting ocurs.

```
ACTION TR-Read( X: data-item, T: Timestamp, var a: value)
{PRE:  $v \in X.V \wedge v.\mathcal{T}_c < \mathcal{T} \wedge v.S = commit \wedge v.value = A$
       $\wedge(\forall x \in X.V - \{v\}, x.S = commit : (x.\mathcal{T}_c < v.\mathcal{T}_c \vee x.\mathcal{T}_c > \mathcal{T}))$
POST:  $a = A$
}
```

Figure 7: TR-Read specification

An additional constraint is generated on the use of data-items by this definition. When data-items can be read and written by both HRT- and SRT-transactions, the following scenario can be envisaged: the HRT-transaction T_H writes to A and reads from A and the SRT-transaction T_S also reads from and writes to A. HRT-transactions must follow a different CCA to read the versions written by T_H from the one to read the version written by T_S. A labelling of A's versions is then necessary. This actually points to a separation of A into a HRT-part and a SRT-part but may lead to badly understood side-effects for the programmer.

Therefore, three sets of objects are defined: (1) SRT-objects only used by SRT-transactions, (2) HRT-objects only used by HRT-transactions and (3) HS-objects used by both types of transactions. The HS-objects are separated into two categories: Hard-Write (HW-)objects and Soft-Write (SW-)objects, where:

HW-object can be written and read by HRT-transactions, but only read by SRT-transactions with the TR-Read action.

SW-object can be written and read by SRT-transactions, but only read by HRT transactions with the TR-Read action.

7.1 Proof of HRT/SRT serializability

The *hrt-graph*, G_{hrt}, is constructed in the same way as the graph G, described in section 6, but includes both HRT and SRT transactions. All edges between HRT-transactions and between SRT-transactions are drawn as specified for the transaction-graph G.

For each HRT-transaction that writes to a set of HW-objects HO, a transaction T_{HO} is imagined that copies all items from HO to another set of objects HO': $T_{HO} = R[HO]\ W[HO']$. The same is done for each SRT-transaction that writes to a set of SW-objects: $T_{SO} = R[SO]\ W[SO']$. T_{SO} and T_{HO} are called *specification transactions.*

The definition of immediately related successor is restricted to transactions of the same class, HRT or SRT. The graph G_{hrt} is extended by adding vertices which represent the above mentioned T_{SO} and T_{HO} and their connecting *transfer-edges.* Edges e labelled with $X \in HW$ with as destination a SRT-transaction T_s are replaced by an edge labelled X from $s(e) = T_h$ to a new HRT-vertex T_{hd} and an edge labelled X' from T_{hd} to $d(e) = T_s$. Edges e labelled with $X \in SW$ with as destination a HRT-transaction T_h are replaced by an edge labelled X from $s(e) = T_s$ to a new SRT-vertex T_{sd} and an edge labelled X' from T_{sd} to $d(e) = T_h$. The $\mathcal{T}_e$ value of a specification transaction, T_{id} is defined as:

$$T_{id}.\mathcal{T}_e = T_i.\mathcal{T}_e \tag{20}$$

The graph G'_{hrt} is constructed from G_{hrt} by adding edges from each vertex T_i to its immediately related successor T_{i+k}, to T_{ix} and to T_{id}, from T_{ix} to T_i's immediately related successor T_{i+k} and from T_{id} to T_i's immediately related successor T_{i+k}.

In Fig. 8 this is visualised for the transactions of T_H and T_S, acting on .

$$\begin{array}{rcl} (hrt:)\,T_H & = & R_H[H]\ R_H[S]\ W_H[H] \\ (srt:)\,T_S & = & R_S[H]\ R_S[S]\ W_S[S] \end{array} \tag{21}$$

The order in which they can be executed by the CCA is:

$$R_S[H_0]\,R_S[S_0]\,R_H[H_0]\,R_H[S_0]\,W_H[H_H]\,W_S[S_S]\,C_H[H,S]\,C_S[H,S] \tag{22}$$

The addition of the specification transaction T_{0d} makes that T_0 is ordered with respect to T_{0d}, T_H and T_S via H and S, T_f is ordered with respect to T_H and T_S via H and S, T_{od} is ordered with respect to T_H and T_S via S' and H', but T_H and T_S are not related and consequently no longer ordered with respect to each other.

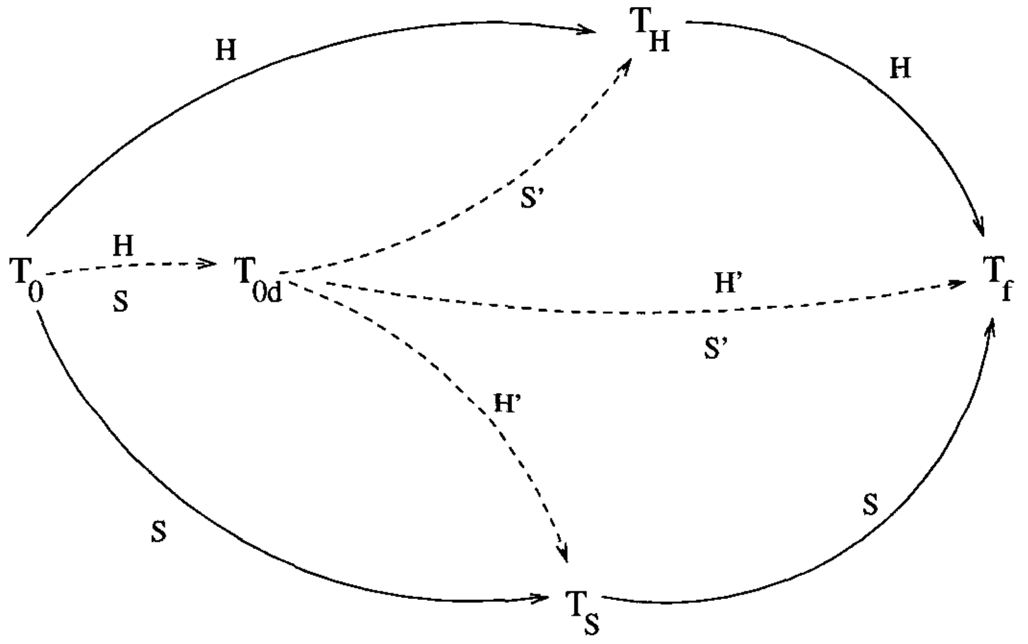
$$\begin{tikzcd} & & T_H \arrow[dr, bend left, "H"] & \\ T_0 \arrow[urr, bend left, "H"] \arrow[r, dashed, "H", "S"'] \arrow[drr, bend right, "S"'] & T_{0d} \arrow[ur, dashed, "S'"] \arrow[rr, dashed, "H'", "S'"'] \arrow[dr, dashed, "H'"] & & T_f \\ & & T_S \arrow[ur, bend right, "S"'] & \end{tikzcd}$$

Figure 8: An example of the graph G_{hrt}

Now the order can be defined:

Definition 4 (hrt-order) *For any two transactions T_i and T_j, $T_i \overset{X}{\prec} T_j$ iff*

- *T_i and T_j are transactions of the same class and ordered according to definition 2.*
- *T_i writes $X \in HO \cup SO$ and $T_j = T_{id}$.*
- *T_j accesses $X \in HO \cup SO$ and T_i is a specification transaction that writes X' and $T_i.\mathcal{T}_e < T_j.\mathcal{T}_s$.*

- *T_i accesses $X \in HO \cup SO$ and T_j is a specification transaction that writes X' and $T_i.\mathcal{T}_s \leq T_j.\mathcal{T}_e$.*

It is important to note that the transfer-serializability concept is independent of the choice of CCA. The SRT-transactions may use a different CCA from the HRT-transacions. The order definition must then be adapted correspondingly.

Lemma 6 *The order of definition 4 is acyclic.*

Proof. Consider the ordering without the specification transactions. Two mutually unordered sets of HRT and SRT transactions exist. According to lemma 3 each individual set is ordered. The addition of the specification transactions can lead to cycles in two ways: (1) when a pair of HRT(SRT)-transactions are differently ordered with respect to each other than with respect to a given specification transaction and (2) when a cyclic ordering can be found from a specification transaction T_{id} via a set of HRT(SRT) transactions to another specification transaction $T_{jd} \neq T_{id}$ and from T_{jd} via a set of SRT(HRT) transactions back to T_{id}.

ad 1) Consider that the following order is valid: $T_j \overset{X'}{\prec} T_{id} \overset{Y'}{\prec} T_k$, with T_j and T_k HRT(SRT)-transactions of the same class. According to definition 4: $T_j.\mathcal{T}_e \leq T_{id}.\mathcal{T}_e < T_k.\mathcal{T}_e$. T_j and T_k write to the same items and conflict. The condition $T_j.\mathcal{T}_e < T_k.\mathcal{T}_e$ implies that T_k committed after T_j. As shown in the proof of lemma 1, this implies that $T_j.C_f < T_k.C_f$. For conflicting T_j and T_k this means that according to definition 2: $T_j \prec T_k$. Consequently, no cycle is possible.

ad 2) Without loss of generality, assume that a chain of HRT(SRT)-transactions is ordered after the specification transaction T_{id} and before the specification transactions T_{jd}. According to the definition 4: $T_{id}.\mathcal{T}_e < T_{jd}.\mathcal{T}_e$. To create a cycle, a chain of SRT(HRT)-transactions is ordered before the specification transaction T_{id} and after the specification transactions T_{jd}. According to the definition 4: $T_{id}.\mathcal{T}_e > T_{jd}.\mathcal{T}_e$. This constitutes a contradiction. Consequently, the order is acyclic. □

For a given transaction, T_r, which uses TR-read and a W-transaction, T_w, of a different class which both access $X \in HO \cup SO$, the following hypothesis, similar to hypothesis 3, is made:

Hypothesis 4 *If T_r reads X at local time t and T_w commits at local time $t' > t$ then $T_r.\mathcal{T}_s < T_w.\mathcal{T}_e$.*

It is proven that the graph G'_{hrt} represents a *transfer-serializable* history for the leaf-transactions which constitute h.

Lemma 7 *For any Y-labelled edge e from G_{hrt}, with $Y = X$ or $Y = X'$: $s(e) \overset{Y}{\prec} d(e)$.*

Proof. When s(e) and d(e) are both HRT- or SRT-transactions, lemma 4 applies and $s(e) \overset{X}{\prec} d(e)$.

One of the two can be a specification transaction. When d(e) is a specification transaction, $s(e) \prec d(e)$ is determined by the hrt-order definition 4. When s(e) is a specification transaction, there exists a T_i such that $s(e) = T_{id}$. When T_i is a HRT(SRT)-transactions then $d(e)$ is a SRT(HRT)-transactions.

The transaction, $d(e)$, only reads X' after the commit of T_i.From the precondition of TR-Read follows: $d(e).\mathcal{T}_s > T_i.\mathcal{T}_e$. From the definition of $\mathcal{T}_e$-value of the specification transaction: $s(e).\mathcal{T}_e < d(e).\mathcal{T}_s$, which according to the definition 4 leads to the conclusion that $s(e) \overset{X'}{\prec} d(e)$.

Combining all cases, it follows that $s(e) \overset{Y}{\prec} d(e)$. □

Theorem 4 (Transfer-serializability) *The graph G'_{hrt} constructed from the ACTION specifications of section 4 and transfer-edges is an acyclic TP(h).*

Proof. For two X-edges e and e' with different sources of which at least one is useful, say e', their destinations are different, $d(e) \neq d(e')$, because a transaction reads a given value only from one transaction at the time. Without loss of generality it is assumed that $s(e) \prec s(e')$. From lemma 7 it is known that $d(e) \succ s(e)$ and $d(e') \succ s(e')$. By construction and by the definition of the transfer-edges, HRT transactions access other data-items than SRT-transactions and are not related. The identical labels of e and e' imply that when one of the four transactions is a HRT(SRT) transaction then the other transactions are either HRT(SRT) transactions or specification transactions.

When $s(e)$, $d(e)$, $s(e')$ and $d(e')$ are all HRT-transactions or SRT-transactions, theorem 2 applies.

Only two cases need to be considered: (1) $d(e)$ is a specification transaction and (2) $s(e')$ is a specification transaction.

Ad (1): According to the definition of transfer transactions, $d(e)$ is ordered before the immediate successor of $s(e)$. $s(e')$ belongs to the same class as $s(e)$ (HRT or SRT) and is the immediate successor of $s(e)$ or $s(e')$ is ordered after the immediate successor and therefore: $d(e) \prec s(e')$.

Ad (2): When $s(e')$ is a specification transaction, $s(e)$ is necessarily also a specification transaction because both edges, e and e' are transfer-edges labeled with the same label X' and only specification transactions are the sources of transfer-edges. There exists a transaction T_i corresponding with the specification transaction $T_{id} = s(e)$ and a transaction T_j corresponding with transaction $T_{jd} = s(e')$. The precondition of TR-Read implies: $T_i.\mathcal{T}_e < d(e).\mathcal{T}_s < T_j.\mathcal{T}_e$ when T_j commits before $d(e)$ reads. Hypothesis 4 assures that the same is true when T_j commits after the read-action of $d(e)$. The definition of the $\mathcal{T}_e$-value of the specification transaction T_{id} states: $T_{id}.\mathcal{T}_e = T_i.\mathcal{T}_e$. Substitution of the definition into the above equation results in: $s(e).\mathcal{T}_e < d(e).\mathcal{T}_s < s(e').\mathcal{T}_e$. According to definition 4: $d(e) \overset{X'}{\prec} s(e')$.

A path can be constructed from T_0 via the acyclic order to $s(e)$, from $s(e)$ via $d(e)$ and $s(e')$ to $d(e')$ and from $d(e')$ via the acyclic order to T_f. Which concludes the proof of the theorem. □

8 Nested atomic objects

A *simple* object that does not invoke any other objects can be considered equivalent to a transaction. The execution of the methods of a set of simple objects should obey the same atomicity requirements as a set of distinct transactions. A *nested* object invokes other objects. Parts of the code of the class separated by invocations of other objects can be identified. A part of a nested object is defined by: (1) it is placed between two object invocations, (2) between the PROCEDURE start and an object invocation and (3) between an object invocation and the PROCEDURE end. The objects invoked by an atomic object should be atomic as well. The concept of atomic nested objects is equivalent to the concept of nested transactions. In Fig. 9, a possible class structure is shown. The PROCEDURE Trans is identified with one transaction T_0. It is divided into three parts (T_1, T_3, T_5) delimited by two object invocations. The graph in the same figure represents the corresponding nested transaction graph. The invoked objects O_1 and O_2 are represented by T_2 and T_4.

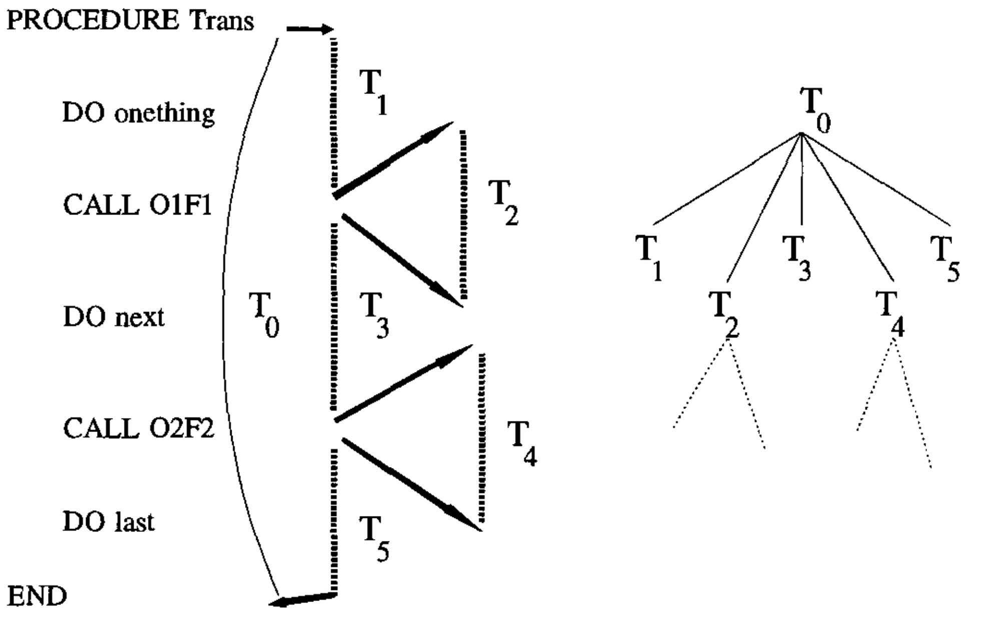

Figure 9: relation between object and nested transaction

The concept of nested transactions was introduced to increase the parallellism inside a transaction and render recovery more efficient [Mos85, Mos87, BBG89]. The individual read and write actions no longer needed to be executed in a fixed serial order, but could be grouped and be executed in parallel. When a subtransaction of a transaction fails, only the subtransaction needs to be recovered.

Although the code in the class should be executed in a serial fashion determined by the language semantics, it is not excluded that object invocations can be done in parallel. When no parallel object invocation are possible, a simplified CCA can be found.

A nested transaction T_i either consists of a sequence of read-actions R_i and write-actions W_i, acting on items $I \in D$ partially ordered in time, or consists of a set of transactions. When the transaction consists of a sequence of actions, it is called a leaf-transaction. When a transaction consists of a set of transactions and is part of another transaction, it is called an intermediate-transaction. A transaction that is not part of any other tranaction is called a root-transaction. A root-transaction is defined to be its own root. The function $root(T)$ returns TRUE iff T is a root-transaction. Fig. 10 illustrates this further. Not shown are leaf-transactions which can be root-transactions at the same time. When a transaction T_i is part of a transaction T_j, T_j is called a parent of T_i, denoted by $T_j = parent(T_i)$. The transaction T_i is an ancestor of T_j, denoted by $T_i \in ancestor(T_j)$, when T_i is a parent of T_j or when T_i is

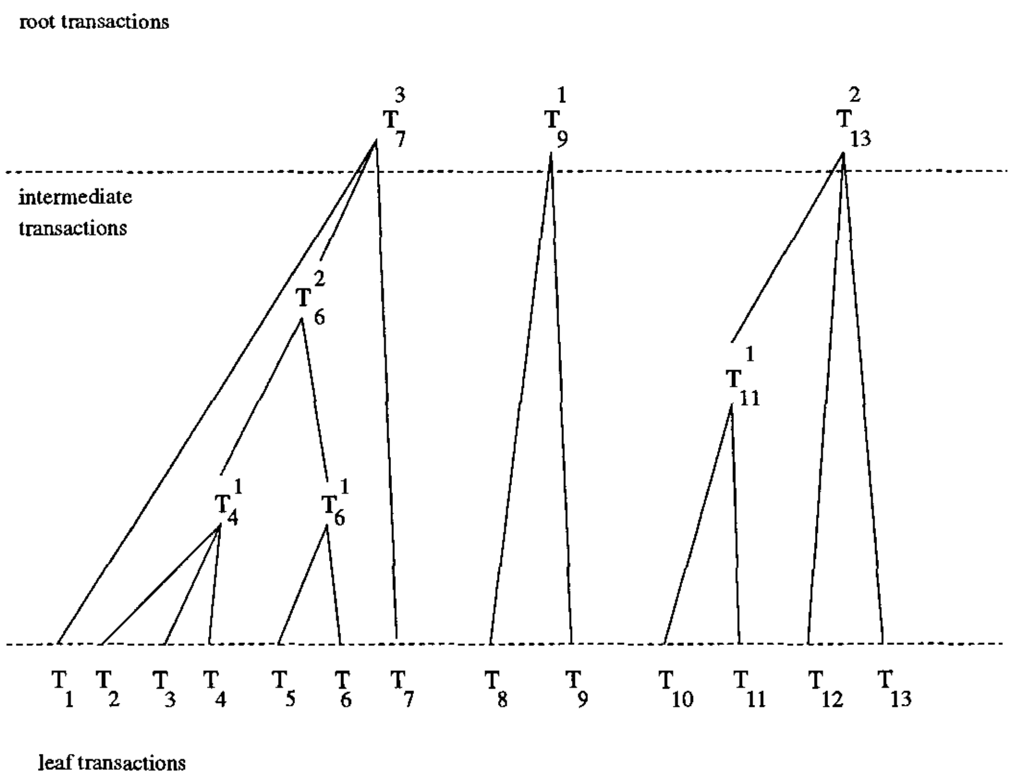
root transactions
intermediate transactions
leaf transactions
T_7^3
T_9^1
T_{13}^2
T_6^2
T_{11}^1
T_4^1
T_6^1
T_1 T_2 T_3 T_4 T_5 T_6 T_7 T_8 T_9 T_{10} T_{11} T_{12} T_{13}

Figure 10: diagram of nested transactions

a parent of another transaction T_k which is an ancestor of T_j. The function *ancestor* has as domain the transactions and as range a set of transactions. The inverse functions are defined by:

$$\begin{array}{c} T_j = parent(T_i) \Leftrightarrow T_i \in child(T_j) \\ T_j \in ancestor(T_i) \Leftrightarrow T_i \in descendant(T_j) \end{array} \tag{23}$$

In the Fig. 10, T_4^1 is an ancestor of $[T_2..T_4]$ and T_7^3 is an ancestor of T_6^2, T_4^1, T_6^1 and $[T_1..T_7]$. The nested structure limits the possible orders in which the leaf-transactions can be executed. For example, the order $[T_{10}, T_{11}, T_{12}, T_{13}]$ in fig. 10 can be changed to $[T_{12}, T_{10}, T_{11}, T_{13}]$ but not to $[T_{10}, T_{12}, T_{11}, T_{13}]$.

A nested transaction T executes a set of transactions denoted by N. N consists of 1 or more transactions T_j. All root transactions are executed such that their result is the same as the result of some serial execution of the same transactions. For a given nested transaction, the transactions $T_i \in N$ are executed such that their result is the same as the result of some serial execution of these same transactions. In Fig 10, a serial execution of root transactions $[T_7^3, T_9^1, T_{13}^2]$ can be realized with the serial execution of the leaf-transactions $[T_1, ..T_{13}]$.

The nesting of transactions has a consequence for the exception atomicity. The results of a leaf-transaction T_i are only visible to another leaf-transaction T_j iff:

- T_i and T_j have different roots and the root of T_i has committed.
- T_i and T_j have a common ancestor T_k which is composed of at least two transactions T_i' and T_j' with $T_i = T_i'$ or $T_i' \in ancestor(T_i)$, $T_j = T_j'$ or $T_j' \in ancestor(T_j)$ and T_i' has

committed.

In Fig. 10, T_8 sees the results of T_5 when T_7^3 has committed and T_5 sees the results of T_3 when T_4^1 has committed.

8.1 Nested transaction algorithm

In Fig. 11, the nested transaction algorithm for OCC-NVTI adapted from Fig. 2 is shown. At the beginning of the nested transaction a timestamp, $\mathcal{T}_s$, as function of the parent timestamp and the current time is created. $\mathcal{T}_s$ uniquely identifies a given transaction. For all transactions of which this transaction is the parent, the leaf or nested procedure is invoked. (see Fig. 2 for leaf-transaction) For all data-items that are accessed by the transaction, a commit is executed.

```
PROCEDURE Nest-Trans(N: transactions, $\mathcal{T}$: timestamp)
        Create timestamp $\mathcal{T}_s = f(\mathcal{T}, time)$
        $\forall T_i \in N$ :    IF $T_i$ is nested THEN Nest-Trans($T_i, \mathcal{T}_s$)
                                 ELSE W-Transaction($T_i.RS, T_i.WS, T_i.I, \mathcal{T}_s$)
                                 FI
        Create timestamp $\mathcal{T}_e = 0$
        Validate($I, \mathcal{T}_s, \mathcal{T}_e, C_f, C_l$)
        IF ($C_f < C_l$) THEN $\forall X \in N.I$: A-Commit( $X, \mathcal{T}_s, \mathcal{T}_e, C_f$ )
        ELSE                  $\forall X \in N.I$: A-Abort( $X, \mathcal{T}_s$)
END
```

Figure 11: Nested transaction algorithm for OCC-NVTI

The visibility of the results within a nested transactions has a consequence for the specification of the actions. Results of intermediate- and leaf-transactions are available to other transactions when they have an ancestor in common. At the commit of a leaf- or intermediate transaction, ownership of its version is passed to its parent; at the commit of the root, the version's state is passed to *commit*. The Read and Commit actions are adapted accordingly. The parameters for Validate and Commit depend on the CCA under consideration. In case of OCC-BV, Validate is only invoked for leaf-transactions. In case of 2PL and MVTSO, Validate is not invoked at all. The actual implementation of the Validate and Commit procedures is discussed in section 10.3.

8.2 Specification of nested transaction actions

Four sets of actions based on 2PL, OCC-BV, OCC-NVTI and MVTSO are specified. All four incorporate versions to allow the ordering of the RO-transactions with respect to the W-transactions such that the values of data-items are available to RO-transactions with minimal waiting times. The read- and write-actions are specified with PRE- and POST-conditions.

The representation of the data-elements contains attributes for all four types of CCAs.

In case of 2PL, a given data-item X is a tuple $< V, L >$, which consists of a set of versions V and a lock L. Each version $x_i \in V$ is a tuple $< \mathcal{T}_w, \mathcal{T}_c, \mathcal{T}_o, S, value >$.

In case of OCC-BV, a given data-item X is a set of versions V. Each version $x_i \in V$ is a tuple $< \mathcal{T}_w, \mathcal{T}_c, \mathcal{T}_o, S, list, value >$.

In case of MVTSO, a given data-item X is a set of versions V. Each version $x_i \in V$ is a tuple $< \mathcal{T}_w, \mathcal{T}_r, \mathcal{T}_c, \mathcal{T}_o, S, value >$.

In case of OCC-NVTI, a given data-item X is a set of versions V. Each version $x_i \in V$ is a tuple $< C_b, C_r, C_e, \mathcal{T}_c, \mathcal{T}_o, S, list, value >$

The elements of the tuples are: (1) $\mathcal{T}_w$ is the timestamp of the writing W-transaction, (2) $\mathcal{T}_r$ is the largest timestamp of all the transactions that have read this version, (3) $\mathcal{T}_c$ is the timestamp created at commit time, (4) $\mathcal{T}_o$ is the timestamp that identifies the owner of the version, (5) C_b is the begin counter of the validity interval defined by the C_f value of the writing transaction, (6) C_r is the maximum of the C_f values of the transaction which read this version, (7) C_e is the end counter of the validity interval, (8) S is the state of the version, (9) *list* is a list of identifiers of transactions that read this version and (10) *value* is the value of the item's version. The state S of a version is (1) *tentative*: the version is created but not yet usable by other transactions than the creating one, (2) *validating*: the transaction that created this version has finished all write and read actions but has not yet asserted their validity and (3) *commit*: the version can be used by transactions with roots different from the creating one. A lock L is a tuple $< s, l >$, which consists of a state L.s and a list L.l of timestamps which uniquely identify the W-transactions using this lock. The lock-state L.s can be in three states: (1) *free*: no W-transaction uses this item, (2) *write*: one W-transaction uses this item for writing and (3) *read*: one or more W-transactions use this item for reading. A lock is only used by W-transactions; RO-transactions don't use the lock but only use the time-stamp $\mathcal{T}_c$ which determines if a version can be read.

```
PROCEDURE RO-Transaction(RS: read actions, I: data-items, t: timestamp)
        Create timestamp $\mathcal{T}_s := f( t, time)$
        $\forall act[X] \in RS$: RO-read(X, $\mathcal{T}_s$, $A_X$)
        execute calculations
END

ACTION RO-Read( X: data-item, $\mathcal{T}$: Timestamp, var a: value)
{PRE:  $v \in X.V \wedge v.\mathcal{T}_c < \mathcal{T} \wedge v.S = commit \wedge v.value = A$
       $\wedge(\forall x \in X.V - \{v\}, x.S = commit : (x.\mathcal{T}_c < v.\mathcal{T}_c \vee x.\mathcal{T}_c > \mathcal{T}))$
POST:  $a = A$
}
```

Figure 12: RO-Read specification and RO-Transaction Algorithm for 2PL and OCC-BV

All the algorithms select a serial order of the transactions, which determines which transactions read the values last modified by a preceding transaction. MVTSO algorithms order with respect to the start time of the transactions, OCC algorithms order with respect to the validation time and 2PL algorithms order with respect to the first lock conflict between any two related transactions.

In Fig. 12, the algorithm for a RO-transaction is shown in case of OCC-BV and 2PL. The specification of the RO-Read states that a version with the largest write timestamp is selected which has a commit timestamp smaller than the timestamp, $\mathcal{T}$, of the reading transaction. For MVTSO (see Fig. 13), the version with the largest write-timestamp $\mathcal{T}_w$ is selected. For OCC-BV and 2PL (see Fig. 12), the version with the largest commit-timestamp $\mathcal{T}_c$ is selected. The value of the selected version is returned. In case of MVTSO, the $\mathcal{T}_r$ values of the versions x with $x.\mathcal{T}_w < \mathcal{T}$ are set equal to the $\mathcal{T}_w$ value of their immediate successor x': $x.\mathcal{T}_r = x'\mathcal{T}_w$.

```
PROCEDURE RO-Transaction(RS: read actions, I: data-items, t: timestamp)
        Create timestamp $\mathcal{T}_s = f(t, time)$
        IF $\exists x \in I : x.\mathcal{T}_c > \mathcal{T}_s$ THEN $\mathcal{T}_m := (\mathbf{min}\ x \in I, x.S \neq Commit \vee x.\mathcal{T}_c > \mathcal{T}_s : x.\mathcal{T}_w)$
        ELSE $\mathcal{T}_m := \mathcal{T}_s$
        $\forall act[X] \in RS$: RO-read($X, \mathcal{T}_m, A_X$)
        execute calculations
END

ACTION RO-Read( X: data-item, $\mathcal{T}$: Timestamp, var a: value)
{PRE:  $v \in X.V \wedge v.\mathcal{T}_w < \mathcal{T} \wedge v.S = commit \wedge v.value = A$
        $\wedge(\forall x \in X.V : (x.\mathcal{T}_w \leq v.\mathcal{T}_w \vee x.\mathcal{T}_w > \mathcal{T}))$
POST:  $a = A \wedge Close(v)$
}
FUNCTION Close( v : version)
{PRE:  $\exists X \in D : v \in X.V \wedge x_m.\mathcal{T}_w = (\mathbf{max}\ x \in X.V, x.\mathcal{T}_w < v.\mathcal{T}_w : x.\mathcal{T}_w) \wedge v.\mathcal{T}_w = T$
POST:  $x_m.\mathcal{T}_r = T \wedge Close(x_m)$
}
```

Figure 13: RO-Read specification and RO-Transaction Algorithm for MVTSO

The case for OCC-VTI is already explained in section 4

8.3 Two Phase locking

The locking algorithm for nested transactions differs from traditional 2PL in the treatment of the locks [Mos85]. Locks set by transactions are inherited by the parents after commit of these transactions. Once a write-lock has been set, transactions can read write-locked data-items as long as the locking transaction is an ancestor of the read-requesting transaction. This strategy assures that when an intermediate transaction aborts, removing intermediate results, all modifications done by one or more transactions that read those intermediate results are also removed. Exception atomicity is maintained inside the nested transaction, because reads on new versions can only take place when these new versions are owned by an ancestor of the reading transaction. A write lock can be set by a leaf-transaction, when all owners of the lock are ancestors of the leaf-transaction. The only owner of the write-lock is the leaf-transaction until its commit.

In Fig. 14, the specifications for the reading, writing and committing of one data-item are shown when the 2PL strategy is used. The last written version (identified by its commit timestamp $\mathcal{T}_c$) is always selected for reading. The reading can start when no tentative version exists and the lock-state is *read* or *free*, or a tentative version is selected of which the owner is an ancestor. After execution of A-Read, the value of v is returned, the reading transaction $\mathcal{T}$ has been added to the lock list and the *free* lock-state is set to *read*.

The writing can only start when the lock-state is *free* or the lock-list is occupied by the invoking transaction or by its ancestors. After execution of A-Write, a version v which contains the specified value is added to the data-item X. The lock-state is set to *write* with $\mathcal{T}$ the only locking transaction specified in the lock-list and the version state is set to *tentative*. The writer and owner of the version identified by respectively $\mathcal{T}_w$ and $\mathcal{T}_o$ are set to $\mathcal{T}$.

The commit is executed on data-items X which are locked by the executing transaction ($\mathcal{T} \in X.L.l$). A-Commit starts with the definition of the set C of versions owned by the committing transaction $\mathcal{T}$. After A-Commit, the executing transaction identifier $\mathcal{T}$ is removed from the lock-list $L.l$ of X. When the root commits, the state of the lock $L.s$ is set to *free*.

```
ACTION A-Read( X: data-item, $\mathcal{T}$: Timestamp, var a: value)
{PRE:    $v \in X.V \wedge X.L.l = L \wedge v.value = A \wedge X.L.s = LS$
         $\wedge(X.L.s \neq write \vee (X.L.s = write \wedge v.\mathcal{T}_o \in ancestor(\mathcal{T})))$
         $\wedge(\forall x \in X.V : v.\mathcal{T}_c \geq x.\mathcal{T}_c)$
POST:    $(LS = free \Rightarrow X.L.s = read) \wedge a = A$
         $\wedge X.L.l = (L \cup \{\mathcal{T}\})$
}

ACTION A-Write( X: data-item, $\mathcal{T}$: Timestamp, a: value)
{PRE:    $(X.L.s = free \vee X.L.l = \{\mathcal{T}\} \vee \forall \mathcal{T}_i \in X.L.l : \mathcal{T}_i \in ancestor(\mathcal{T}))$
         $\wedge\ X.V = Q \wedge a{=}A$
POST:    $v$ = newly created version $\wedge X.V = Q \cup \{v\}$
         $\wedge v.S = tentative \wedge v.value = A \wedge v.\mathcal{T}_o = \mathcal{T} \wedge v.\mathcal{T}_w = \mathcal{T}$
         $\wedge X.L.s = write \wedge X.L.l = \{\mathcal{T}\}$
}

ACTION A-Commit( X: data-item, $\mathcal{T}$: Timestamp, t: Timestamp)
{PRE:    $X.L.l = L \wedge X.V = Q \wedge \mathcal{T} \in L$
         $\wedge C = \{v \mid v \in Q \wedge v.\mathcal{T}_o = \mathcal{T}\}$
POST:    $(\forall v \in C : (v.\mathcal{T}_w = \mathcal{T} \Rightarrow v.\mathcal{T}_c = t)$
            $\wedge(root(\mathcal{T}) \Rightarrow (v.S = commit \wedge X.L.l = \emptyset \wedge X.L.s = free))$
            $\wedge(\neg root(\mathcal{T}) \Rightarrow v.\mathcal{T}_o = parent(\mathcal{T}) \wedge X.L.l = (L - \{\mathcal{T}\}) \cup \{parent(\mathcal{T})\})$
}

ACTION A-Abort( X: data-item, $\mathcal{T}$: Timestamp)
{PRE:    $X.V = Q \wedge C = \{v \mid v \in X.V \wedge v.\mathcal{T}_o = \mathcal{T}\}$
POST:    $X.V = Q\text{-}C$
}
```

Figure 14: 2PL read, write and commit specifications

When the transaction is not the root, $\mathcal{T}$'s parent is added to the lock-list and the owners of all versions of C are set to the parent of $\mathcal{T}$. The Commit times are set to t for all leaf-transactions which wrote this version ($\mathcal{T}_w = \mathcal{T}$).

The A-Abort action specifies that the tentative versions owned by this transaction are removed.

8.4 Multi-Version Timestamp ordering

MVTSO assures that for a set of transactions, the order of the equivalent serialized set is equal to the order in which the transactions are started. For nested transactions, the order of the root-transactions is determined by the absolute start times of the root-transactions, but the order within a nested transaction is determined by the relative start times of the intermediate transactions. As for 2PL, transactions can only read values from versions committed by earlier root transactions or from versions which are owned by ancestors. Each version has a write timestamp of the writing transactions and a read timestamp that is the maximum of the timestamps of all transactions that read this version. A serializable schedule is enforced by the simple rule that for any two versions u and v with write-timestamps $\mathcal{T}_u < \mathcal{T}_v$ and v the immediate successor of u, the timestamps $\mathcal{T}_r$ of the transactions that read version u obey: $\mathcal{T}_u \leq \mathcal{T}_r < \mathcal{T}_v$.

In Fig. 15, the specifications for the reading, writing and committing of one data-item are shown when the MVTSO strategy is used. The reading can start when there is a version, committed by an ancestor (identified with $v.\mathcal{T}_o \in ancestor(\mathcal{T})$) or by a root transaction, with

```
ACTION A-Read( X: data-item, T: Timestamp, var a: value)
{PRE:    $v \in X.V \wedge v.\mathcal{T}_r = TR \wedge v.\mathcal{T}_w \leq \mathcal{T} \wedge v.value = A \wedge (v.S = commit \vee v.\mathcal{T}_o \in ancestor(\mathcal{T}))$
         $\wedge(\forall x \in X.V - \{v\} : (x.\mathcal{T}_w < v.\mathcal{T}_w \vee x.\mathcal{T}_w > \mathcal{T}))$
POST:    $v.\mathcal{T}_r = max(TR, \mathcal{T}) \wedge a = A$
}

ACTION A-Write( X: data-item, T: Timestamp, a: value)
{PRE:    $X.V = Q \wedge a = A \wedge (\forall x \in Q : (x.\mathcal{T}_r < \mathcal{T}) \vee x.\mathcal{T}_w > \mathcal{T})$
POST:    $v$ = newly created version
         $\wedge v.\mathcal{T}_w = \mathcal{T} \wedge v.\mathcal{T}_o = \mathcal{T} \wedge v.\mathcal{T}_r = \mathcal{T} \wedge v.S = tentative$
         $\wedge v.value = A \wedge X.V = Q \cup \{v\}$
}

ACTION A-Commit( X: data-item, T: Timestamp, t: Timestamp)
{PRE:    $C = \{v \mid v \in X.V \wedge \mathcal{T} = v.\mathcal{T}_o\}$
POST:    $\forall v \in C : (v.\mathcal{T}_w = \mathcal{T} \Rightarrow v.\mathcal{T}_c = t)$
             $\wedge(\ (root(\mathcal{T}) \Rightarrow v.S = commit)$
             $\wedge(\neg root(\mathcal{T}) \Rightarrow v.\mathcal{T}_o = parent(\mathcal{T}))\ )$
}

ACTION A-Abort( X: data-item, T: Timestamp)
{PRE:    $X.V = Q \wedge C = \{v \mid v \in X.V \wedge \mathcal{T} = v.\mathcal{T}_o\}$
POST:    $X.V = Q\text{-}C$
}
```

Figure 15: MVTSO read, write and commit specifications

a write-timestamp smaller than the timestamp of the executing transaction. The version v with the largest write-timestamp is selected. After execution of A-Read, the read-timestamp is updated with the timestamp of the executing transaction when this timestamp is larger than the original read-timestamp. The value of v is returned in a.

The writing can start when there is no version with a write-timestamp smaller and a read-timestamp larger than the timestamp of the executing transaction. After execution of A-Write, a new version is created with the specified value and a write-timestamp, owner-timestamp and read-timestamp equal to the timestamp of the executing transaction. The state of the version is set to *tentative*.

A-Commit is executed to make the versions accessible to other (root) transactions and their descendants. The set C contains the versions owned by the committing transaction. After A-Commit, the owners of the versions are equal to the parent of the committing transaction. In case of a leaf-transaction ($v.\mathcal{T}_w = \mathcal{T}$), the commit timestamps of these versions are set equal to the specified commit timestamp. For root transactions, the states of C's versions are set to *commit*.

The A-Abort action specifies that all versions owned by the executing transaction are removed.

8.5 OCC-BV

In Optimistic Concurrency Control with Backward Validation (OCC-BV), the results of all write actions of a given transaction are installed in one atomic step including a "validate" phase. Before, all write actions have taken place without any checks on their validity. The ordering of the serialized equivalent set of transactions is the same as the one dictated by the commit times. At validation time, each transaction verifies that no more recent version is installed since its reading of the corresponding item. When new versions are installed, the transaction should have read this newer version. Aborts of transactions are then necessary.

```
ACTION A-Read( X: data-item, T: Timestamp, var a: value)
{PRE:   $v \in X.V \wedge (v.S = commit \vee v.\mathcal{T}_o \in ancestor(\mathcal{T})) \wedge v.list = L \wedge v.value = A$
        $\wedge(\forall x \in X.V - \{v\} : (x.S = commit \vee x.\mathcal{T}_o \in ancestor(\mathcal{T})) \Rightarrow v.\mathcal{T}_c > x.\mathcal{T}_c)$
POST:   $v.list = L \cup \{T\} \wedge a = A$
}

ACTION A-Write( X: data-item, T: Timestamp, a: value)
{PRE:   $X.V = Q \wedge a = A$
POST:   $v$ = newly created version
        $\wedge v.S = tentative \wedge v.\mathcal{T}_o = \mathcal{T} \wedge v.\mathcal{T}_r = \mathcal{T} \wedge v.value = A \wedge X.V = Q \cup \{v\}$
}

ACTION A-Validate(X: data-item, T: Timestamp, t: Timestamp)
{PRE:   $(\forall x \in X.V : x.S \neq validating) \wedge T_L = (\mathbf{max}\ x \in X.V, (x.S = commit \vee x.\mathcal{T}_o \in ancestor(\mathcal{T})) : x.\mathcal{T}_c)$
        $\wedge(\forall x \in X.V : x.\mathcal{T}_c \neq T_L \Rightarrow \mathcal{T} \notin x.list)$
POST:   $\forall v \in X.V, v.\mathcal{T}_o = \mathcal{T} : v.S = validating$
        $\wedge(\exists v \in X.V : v.\mathcal{T}_o = \mathcal{T}) \Rightarrow t = time \circ pid$
        $\wedge(\forall v \in X.V : v.\mathcal{T}_o \neq \mathcal{T}) \Rightarrow t = T_L \circ pid$
}

ACTION A-Commit( X: data-item, T: Timestamp, t: Timestamp)
{PRE:   $C = \{x \mid x \in X.V \wedge x.\mathcal{T}_o = \mathcal{T}\} \wedge \forall x \in C : x.S = ST_x$
        $\wedge \forall x \in X.V - C : (x.\mathcal{T}_c < (\mathbf{min}\ v \in C : v.\mathcal{T}_c)) \vee (x.\mathcal{T}_c > (\mathbf{max}\ v \in C : v.\mathcal{T}_c))$
POST:   $\forall x \in C : (\ (ST_x = validating \Rightarrow x.\mathcal{T}_c = t)$
            $\wedge((root(\mathcal{T}) \wedge x.S = commit) \vee (\neg root(\mathcal{T}) \wedge x.S = tentative \wedge x.\mathcal{T}_o = parent(\mathcal{T})))\ )$
}

ACTION A-Abort( X: data-item, T: Timestamp)
{PRE:   $X.V = Q \wedge C = \{v \mid v \in X.V \wedge \mathcal{T} = v.\mathcal{T}_o)\}$
POST:   $X.V = Q\text{-}C$
}
```

Figure 16: OCC-BV read, write, validate and commit specifications

In Fig. 16, the specifications for the reading, writing, validating, and committing of one data-item are shown when the OCC-BV strategy is used. The reading can start by selecting a committed version or a tentative version owned by an ancestor of $\mathcal{T}$ (identified by $(v.\mathcal{T}_o \in ancestor(\mathcal{T}))$. The version v with the highest commit-timestamp is selected. After execution of A-Read, the value of v is returned and the timestamp of the executing transaction is added to the version's list.

The writing can always start. A new version with a *tentative* state and the specified value is created. The owner is set to be the writing leaf-transaction.

Only leaf-transactions validate their results. Validation can only start when no other versions are being validated. After execution of A-Validate, the state of the version written by this transaction, $v.\mathcal{T}_o = \mathcal{T}$, is set to *validating*. The read-actions of the transaction are valid when at validation time there are no new versions created since the reading of the data-items. This is verified by looking at the lists of transactions that read a given version. Only the last validated version (defined by T_L) is allowed to be read by the validating transaction. A value for the local timestamp of $\mathcal{T}_e$ is returned in t.

Each item X is represented by a list of versions ordered by the commit times of the leaf-transactions. A nested transaction $\mathcal{T}$ can own one or more versions of an item. All versions not belonging to $\mathcal{T}$ must be ordered either before all versions of $\mathcal{T}$ or after. Consequently, A-Commit can only start when all versions created by leaf-transactions of the committing transaction, $\mathcal{T}$, constitute one contiguous sequence. A-Commit is executed on all versions which are created by the committing transaction. When leaf-transactions ($v.S = validating$) are committed, the commit timestamp is replaced by the specified one. In case of a root transaction, the version's state is set to *commit*; in the other case, the state is set to *tentative* and the owner-timestamp is set equal to the timestamp of the parent of the comitting transaction. In case this data-item was read, the reading timestamp is set equal to the maximum of the committing transaction and former read-timestamp.

The A-Abort action specifies that all versions owned by the executing transaction are removed.

8.6 OCC-NVTI

```
ACTION A-Read( X: data-item, $\mathcal{T}_i$: Timestamp, var a: value)
{PRE:    $v \in X.V \wedge (v.S = commit \vee v.\mathcal{T}_o \in ancestor(\mathcal{T}_i)) \wedge v.list = L \wedge v.value = A$
         $\wedge(\forall x \in X.V, (x.S = commit \vee x.\mathcal{T}_o \in ancestor(\mathcal{T}_i)) :\ x.\mathcal{T}_c \leq v.\mathcal{T}_c)$
POST:    $v.list = L \cup \{\mathcal{T}_i\} \wedge a = A$
}

ACTION A-Write( X: data-item, $\mathcal{T}_i$: Timestamp, var $C_f$: counter, a: value)
{PRE:    $X.V = Q \wedge a = A$
POST:    $v$ = newly created version
         $\wedge v.S = tentative \wedge v.\mathcal{T}_o = \mathcal{T}_i \wedge v.value = A \wedge v.list = \emptyset \wedge X.V = Q \cup \{v\}$
}

ACTION A-Validate(X: data-item, $\mathcal{T}_i$, t: Time-stamp, var $C_f$,$C_l$: Counter)
{PRE:    $\forall x \in X.V : x.S \neq validating$
         $P(x) \equiv (\neg root(\mathcal{T}_i) \Rightarrow x.\mathcal{T}_o = parent(\mathcal{T}_i)) \wedge (root(\mathcal{T}_i) \Rightarrow x.S = commit)$
POST:    $(\forall x \in X.V, x.\mathcal{T}_o = \mathcal{T}_i : x.S = validating)$
         $\wedge((\exists v \in X.V : v.\mathcal{T}_o = \mathcal{T}_i) \Rightarrow C_f = (\max x \in X.V, P(x) : x.C_r + 1))$
         $\wedge((\forall v \in X.V : v.\mathcal{T}_o \neq \mathcal{T}_i) \Rightarrow C_f = (\max x \in X.V, P(x) \wedge \mathcal{T}_i \in x.list : x.C_b + 1))$
         $\wedge((\exists v \in X.V, P(x) : \mathcal{T}_i \in v.list) \Rightarrow C_l = v.C_e - 1)$
         $\wedge((\forall v \in X.V : \mathcal{T}_i \notin v.list) \Rightarrow C_l = \infty)$
         $\wedge(\exists v \in X.V : v.\mathcal{T}_o = \mathcal{T}) \Rightarrow t = time \circ pid$
         $\wedge(\forall v \in X.V : v.\mathcal{T}_o \neq \mathcal{T}) \Rightarrow t = (\max x \in X.V, P(x) : x.\mathcal{T}_c) \circ pid$
}

ACTION A-Commit( X: data-item, $\mathcal{T}_i$,$\mathcal{T}_e$: Time-stamp, $C_f$: Counter)
{PRE:    $(\forall v \in X.V : v.C_r = C_v \wedge v.list = L_v \wedge v.\mathcal{T}_o = \mathcal{T}_{ov})$
         $P(x) \equiv (\neg root(\mathcal{T}_i) \Rightarrow x.\mathcal{T}_o = parent(\mathcal{T}_i)) \wedge (root(\mathcal{T}_i) \Rightarrow x.S = commit)$
         $\wedge b \in X.V \wedge P(b) \wedge (\forall x \in X.V, P(x) : x.C_b \leq b.C_b)$
POST:    $((\exists x \in X.V : \mathcal{T}_{ox} = \mathcal{T}_i) \Rightarrow b.C_e = C_f)$
         $\wedge(\forall v \in X.V, \mathcal{T}_{ov} = \mathcal{T}_i : v.C_e = \infty \wedge v.C_b = C_f \wedge v.C_r = C_f \wedge v.\mathcal{T}_c = \mathcal{T}_e)$
         $\wedge(\forall v \in X.V, \mathcal{T}_i \in L_v : (P(v) \Rightarrow v.C_r = \max(C_v, C_f)) \wedge (\neg P(v) \Rightarrow v.list = (L_v \setminus \mathcal{T}_i) \cup parent(\mathcal{T}_i)))$
         $\wedge(\forall v \in X.V, \mathcal{T}_{ov} = \mathcal{T}_i : (root(\mathcal{T}_i) \Rightarrow v.S = commit) \wedge (\neg root(\mathcal{T}_i) \Rightarrow v.\mathcal{T}_o = parent(\mathcal{T}_i)))$
}

ACTION A-Abort( X: data-item, $\mathcal{T}_i$: Timestamp)
{PRE:    $X.V = Q \wedge C = \{v \mid v \in X.V \wedge \mathcal{T}_i = v.\mathcal{T}_o)\}$
POST:    $X.V = Q\text{-}C$
}
```

Figure 17: nested OCC-NVTI read, write, validate, commit and abort specifications

Optimistic Conncurrency Control with Nested Versions and Time Intervals (OCC-NVTI) is based on OCC-VTI. Nesting is added by ordering all transactions with the same parent according to the C_f values. Recursive application of the ordering until the roots are committed results in root transactions which are ordered like the unnested OCC-VTI transactions.

Readable versions are the committed versions and the versions which are owned by an ancestor of the reading transactions. Reading takes place by selecting from all readable versions the last written one which has the largest $\mathcal{T}_c$ value. After reading, the value of the version is returned to the invoking transaction and the identifier of the reading transaction is added to the read-list of the selected version.

Writing can always take place. The version is added to the list of versions with state *tentative.*

When A-Validate starts, no other transactions can be validating the same item. For every item the valid interval is determined. Two cases are considered: (1) the transactions is a root-transaction and (2) the transaction is not a root-transaction. In case 2, the versions belonging to the parent transaction and in case 1, the committed versions are considered. Additionally the read versions are considered by looking at versions with a list containing the identifier of the validating transaction. The returned C_f value is the maximum of all version's C_b; the returned C_l value is the minimum of all C_r values.

A-Commit adjusts the C_b, C_r and C_e values of the versions and the involved lists. When the Committing transaction is the root, the version's state is set to *commit.* When not a root, the owner is set to the parent of the committing transaction. The last readable version is identified and the C_f values in the readers list are set equal to the C_b value of the currently committing transaction. The C_f values of the transactions in the readers list of the versions owned by the committing transaction are set equal to $C_b + 1$.

```
ACTION A-Commit( X: data-item, $\mathcal{T}_i$,$\mathcal{T}_e$: Timestamp, $C_f$: Counter)
{PRE:     $X.V = Q \wedge Z = \{v \mid v \in X.V \wedge \mathcal{T}_i = v.\mathcal{T}_o)\} \wedge C_{max} = (\max x \in Z : x.C_b)$
          $R = \{v \mid v \in Z \wedge v.C_b < C_{max}\}$
POST:     $X.V = Q\text{-}R$
}
```

Figure 18: specification of version removal in A-Commit for OCC-NVTI

Only the latest version of a set of versions owned by the same transaction can be read. All versions owned by the same transaction have the same C_b value which leads to ambiguities. Therefore, only the version with the largest C_b value will be retained; all others are removed (see Fig. 18).

8.7 Commit

The precedence relation together with view serializability make that when for a given data-item several versions are committed by one root-transaction, only one of those versions is actually read by other root-transactions or their descendants. To diminish the number of versions, all unused versions should be removed. The commit timestamp $\mathcal{T}_c$ of the last version has been modified to the current time plus some constant d. The post conditions of commit have been extended as shown:

$$\begin{array}{llll} & \text{OCC-BV, 2PL} & \text{MVTSO} & \\ PRE: & v.\mathcal{T}_c = (\mathbf{max}\ x \in C : x.\mathcal{T}_c) & v.\mathcal{T}_w = (\mathbf{max}\ x \in C : x.\mathcal{T}_w) & \\ & t = timestamp(time) + d & t = timestamp(time) + d & \\ & \wedge R = \{x \mid x \in C \wedge x \neq v\} & \wedge R = \{x \mid x \in C \wedge x \neq v\} & (24) \\ POST: & root(\mathcal{T}) \Rightarrow X.V = Q - R & root(\mathcal{T}) \Rightarrow X.V = Q - R & \\ & v.\mathcal{T}_c = t & v.\mathcal{T}_c = t & \end{array}$$

9 Proof of correctness

For the proof of OCC-BV, OCC-NVTI and 2PL, the following hypothesis relates the commit timestamps of the transactions with the order in which the transactions execute the commit actions on the individual data-items.

Hypothesis 5 *In case of 2PL, OCC-BV and OCC-NVTI, two RW-conflicting transactions T and T', with commit-timestamps $\mathcal{T}_e$ and $\mathcal{T}'_e$, execute their commit-actions at local times t and t' such that: $t > t' \Leftrightarrow \mathcal{T}_e > \mathcal{T}'_e$.*

A last hypothesis on the implementation of timestamps must be stated for MVTSO.

Hypothesis 6 *$(T_i.\mathcal{T}_s < parent(T_k).\mathcal{T}_s < T_j.\mathcal{T}_s \wedge T_j \notin descendant(parent(T_k)) \wedge parent(T_k) \notin descendant(T_i)) \Rightarrow T_i.\mathcal{T}_s < T_k.\mathcal{T}_s < T_j.\mathcal{T}_s$*

RO-transactions read their data from one or more W-transactions or T_0. For a given RO-transaction, T_{ro}, a set of versions exists associated with the read set RS_{ro}. In case of MVTSO, the maximum write-timestamp $\mathcal{M}_{r0}$ is defined as the minimum of all $\mathcal{T}_w$ for which the versions have a commit-timestamp $\mathcal{T}_c > T_{ro}.\mathcal{T}_s$ when there are such versions:

$$\mathcal{M}_{ro} = (\mathbf{min}\ x \in RS_{ro}, (x.\mathcal{T}_c > T_{ro}.\mathcal{T}_s \vee x.S \neq commit) : x.\mathcal{T}_w) \tag{25}$$

Otherwise: $\mathcal{M}_{ro} = \mathcal{T}_s$. The X-order, $\overset{X}{\prec}$, can be extended for MVTSO, OCC-BV and 2PL.

Definition 5 (Nested X-orderl) *For any two* **leaf**-*transactions T_i and $T_j, T_i \overset{X}{\prec} T_j$ iff*

- *T_i and T_j are conflicting W-transactions with $X \in RS_i \cap WS_j \vee X \in WS_i \cap RS_j$ and $T_i.T_y < T_j.T_y$, where $T_y = \mathcal{T}_e$ for OCC-BV & 2PL and $T_y = \mathcal{T}_s$ for MVTSO.*
- *$(i = 0 \wedge X \in RS_j)$ or $(j = f \wedge X \in WS_i)$.*
- *T_i is a W-transaction and T_j is a RO-transaction and $X \in WS_i \cap RS_j$ and for OCC-BV & 2PL: $T_i.\mathcal{T}_e < T_j.\mathcal{T}_s$ and for MVTSO: $T_i.\mathcal{T}_s < \mathcal{M}_j$.*
- *T_j is a W-transaction and T_i is a RO-transaction and $X \in WS_j \cap RS_i$ and for OCC-BV & 2PL: $T_i.\mathcal{T}_s < T_j.\mathcal{T}_e$ and for MVTSO: $\mathcal{M}_i \leq T_j.\mathcal{T}_s$.*

A different order is defined in the case of OCC-NVTI. The root of transaction T_i is denoted with T_i^*.

Definition 6 (Nested-X-order2) *For any two transactions T_i and T_j, conflicting in X, $T_i \overset{X}{\prec} T_j$ iff*

- *T_i and T_j are root-transactions and ordered according to definition 2*
- *T_i is a root-transaction and T_j is not a root-transactions and $T_j^*.C_f > T_i.C_f$*
- *T_j is a root-transaction and T_i is not a root-transactions and $T_i^*.C_f < T_j.C_f$*
- *T_j and T_i are not root-transactions and $T_j^*.C_f > T_i^*.C_f$*
- *T_i and T_j have the same root and there is a transaction T_k with $T_i, T_j \in descendant(T_k)$ with children T_m and T_l such that $T_i = T_m \vee T_i \in descendant(T_m)$ and $T_j = T_l \vee T_j \in descendant(T_l)$ and $T_m.C_f < T_l.C_f$.*

The consistency of these relations is proven in the following lemma:

Lemma 8 *For two related transactions T_i and T_j, with $\{X,Y\} \subseteq (RS_i \cap WS_j) \cup (RS_j \cap WS_i)$: $T_i \overset{X}{\prec} T_j \Leftrightarrow T_i \overset{Y}{\prec} T_j$*

Proof. When both transaction are W-transactions, they are ordered by the same timestamps independent of the variable with which they are related. Consequently, the lemma is true for W-transactions.

The rest of the proof is identical to the proof of lemma 2.

□

Lemma 9 *The order of definition 5 is acyclic.*

Proof. The W-transactions are ordered by their uniquely attributed timestamps. The total ordering of the timestamps assures that the set of W-transactions is ordered without cycles. In case of 2PL and OCC-BV the ordering criterium of RO-transactions and W-transactions is the same (defined by $\mathcal{T}_e$) and the union of both sets is also ordered without cycles. The case of MVTSO must be considered separately. No order is defined between RO-transactions, but RO-transactions are ordered with respect to W-transactions in two ways. Consequently, cycles can occur when a RO-transaction T_{ro} is ordered between two W-transactions, T_{w1} and T_{w2}, such that $T_{w1} \overset{Z}{\prec} T_{w2} \wedge T_{w2} \overset{X}{\prec} T_{ro} \overset{Y}{\prec} T_{w1}$, as shown in fig 5. From definition 2, the following relations can be derived:

$$T_{w2} \overset{X}{\prec} T_{ro} \Rightarrow T_{w2}.\mathcal{T}_s < \mathcal{M}_{ro} \tag{26}$$

$$T_{ro} \overset{Y}{\prec} T_{w1} \Rightarrow \mathcal{M}_{ro} \leq T_{w1}.\mathcal{T}_s \tag{27}$$

Combining both relations leads to: $T_{w1}.\mathcal{T}_s > T_{w2}.\mathcal{T}_s$. Because $(Z \in RS_{w1} \cap WS_{w2}) \vee (Z \in WS_{w1} \cap RS_{w2})$, it follows that $T_{w2} \overset{Z}{\prec} T_{w1}$ and the cycle is impossible. □

The same needs to be proven for the order definition 6 of OCC-NVTI:

Lemma 10 *The order of definition 6 is acyclic.*

such a version is commited by either a root transaction or a leaf transaction. It follows that v is committed by the writing leaf-transactions and one or more ancestors when $d(e)$ reads v. According to the local clock of X, a reading leaf-transaction commits after the commit of the leaf-transactions that wrote the version. According to hypothesis 5: $s(e).\mathcal{T}_e < d(e).\mathcal{T}_e$.

MVTSO A-Read proceeds by reading a version, v, with its state *commit* or by reading the tentative version, v, owned by an ancestor of the reading transaction.

$$v.S = commit \vee v.\mathcal{T}_o \in ancestor(\mathcal{T}) \tag{31}$$

The reading MVTSO leaf-transaction reads versions such that $s(e).\mathcal{T}_s < d(e).\mathcal{T}_s$, as prescribed by the precondtion of the A-Read action:

$$v.\mathcal{T}_w \leq \mathcal{T}_s \tag{32}$$

with $v.\mathcal{T}_w = s(e).\mathcal{T}_s$ and $\mathcal{T}_s = d(e).\mathcal{T}_s$.

OCC-BV A-Read proceeds by reading a version with its state *commit* or by reading the tentative version owned by an ancestor of the reading transaction.

$$v.S = commit \vee v.\mathcal{T}_o \in ancestor(\mathcal{T}) \tag{33}$$

According to the post condition of A-Commit

$$(root(\mathcal{T}) \Rightarrow (v.S = commit)) \wedge (\neg root(\mathcal{T}) \Rightarrow (v.\mathcal{T}_o = parent(\mathcal{T})) \tag{34}$$

such a version is committed by either a root transaction or a leaf transaction. According to the local clock of X, the reading leaf-transaction commits after the commit of the leaf-transactions that wrote the version. According to hypothesis 5: $s(e).\mathcal{T}_e < d(e).\mathcal{T}_e$.

OCC-NVTI Assume that leaf-transaction T_i has written a version v read by leaf-transaction T_j. According to the precondition of A-Read

$$v \in X.V \wedge (v.S = commit \vee v.\mathcal{T}_o \in ancestor(\mathcal{T}_j)) \tag{35}$$

T_j can read either a committed version or a version owned by an ancestor.

Assume the version is owned by an ancestor. According to the postcondition of A-Commit

$$\neg root(\mathcal{T}_i) \Rightarrow v.\mathcal{T}_o = parent(\mathcal{T}_i) \tag{36}$$

the ownership of v passes to the parent of T_i. At the commit of the parent or ancestor of T_i, ownership passes to another ancestor of T_i. When T_j reads v, the owner T_k of v is both an ancestor of T_j and T_i. The postcondition of A-Read specifies that after the reading of v by T_j, an entry $\mathcal{T}_j$ is added to $v.list$. According to the postcondition of A-Commit

$$\text{PRE:} \quad \forall v \in X.V : v.list = L_v \tag{37}$$

$$P(x) \equiv (\neg root(\mathcal{T}_i) \Rightarrow x.\mathcal{T}_o = parent(\mathcal{T}_i)) \tag{38}$$

$$\text{POST:} \quad \neg P(v) \Rightarrow v.list = (L_v \setminus \mathcal{T}_j) \cup parent(\mathcal{T}_j)) \tag{39}$$

Proof. The ordering between RO-transactions and W-transactions is already proven in lemma 3. Consider the case of ordered nested W-transactions with $T_i \overset{X}{\prec} T_j \overset{Y}{\prec} T_k$. Assume that T_i and T_k conflict in Z. When all transactions are root-transactions: $T_i.C_f < T_j.C_f < T_k.C_f$. According to definition 6: $T_i \overset{Z}{\prec} T_k$.

Assume T_i and T_j have different roots: $T_i^* \neq T_j^*$. According to the nested order definition: $T_i^*.C_f < T_j^*.C_f$ and $T_j^*.C_f \leq T_k^*.C_f$. Combining both equations leads to $T_i^*.C_f < T_k^*.C_f$. According to the order definition it follows that $T_i \overset{Z}{\prec} T_k$.

Assume T_i and T_j have a common ancestor T_l; T_j and T_k have a common ancestor T_m. Then there are two children T_{l1} and T_{l2} of T_l with $T_i = T_{l1} \vee T_i \in \mathit{descendant}(T_{l1})$ and $T_j = T_{l2} \vee T_j \in \mathit{descendant}(T_{l2})$ and $T_{l1}.C_f < T_{l2}.C_f$. There are also two children T_{m1} and T_{m2} of T_m with $T_j = T_{m1} \vee T_j \in \mathit{descendant}(T_{m1})$ and $T_k = T_{m2} \vee T_k \in \mathit{descendant}(T_{m2})$ and $T_{m1}.C_f < T_{m2}.C_f$. If $T_{m1} = T_{l2}$ then it follows immediately that $T_i \overset{X}{\prec} T_k$. If $T_{m1} \neq T_{l2}$ then $T_m \in \mathit{ancestor}(T_l) \vee T_l \in \mathit{ancestor}(T_m)$. Both cases are treated in the same way. Consider $T_m \in \mathit{ancestor}(T_l)$. Then either $T_{m1} = T_l$ or $T_{m1} \in \mathit{ancestor}(T_l)$. From the definition of T_{l1} it follows that $T_{m1} \in \mathit{ancestor}(T_i)$. From the order definition, $T_k \in \mathit{ancestor}(T_{m2})$ and $T_{m1}.C_f < T_{m2}.C_f$ it follows that $T_i \overset{Z}{\prec} T_k$. □

Based on order definitions 6 and 5 the view-serializability of the nested CCA's can be proven. It is first demonstrated that the history of leaf-transactions is serializable.

Lemma 11 *For any labelled edge e from G, with $s(e) = T_i$ and $d(e) = T_j$, T_i and T_j both leaf-transactions related via X, it holds that $T_i \overset{X}{\prec} T_j$.*

Proof. A RO-transaction T_{ro} is the destination of a given edge e, or is the source of an edge with destination T_f. In the first case, the pre-condition in RO-Read and the assignment to the parameters, $\mathcal{T}$ or C , of RO-Read in RO-Transaction:

$$\begin{array}{ll} \text{2PL \& OCC-BV} & v.\mathcal{T}_c < \mathcal{T} \wedge v.S = \mathit{commit} \wedge \mathcal{T} = \mathcal{T}_s \\ \text{MVTSO} & v.\mathcal{T}_w < \mathcal{T} \wedge v.S = \mathit{commit} \wedge \mathcal{T} = \mathcal{M}_{ro} \\ \text{OCC-NVTI} & v.C_b < C \wedge v.S = \mathit{commit} \wedge C = \mathcal{M}_{ro} \end{array} \tag{28}$$

assure that $s(e).\mathcal{T}_e < d(e).\mathcal{T}_s$ in case of OCC-BV & 2PL, $s(e).\mathcal{T}_s < \mathcal{M}_{ro}$ for MVTSO and $s(e).C_f < \mathcal{M}_{ro}$ for OCC-NVTI. For all four CCA's, $s(e)$ precedes $d(e)$ in the defined order.

In the second case, T_f follows all other transactions per definition. Consequently, the lemma is true for all edges involving a RO-transaction. The same reasoning is valid for any edge involving $T_{ix} \in H^u$.

Edges which only involve W-transactions are considered for all four types of CCA separately.

2PL A-Read proceeds by reading the item that is free, read-locked or write-locked but owned by a parent:

$$X.L.s \neq \mathit{write} \vee (X.L.s = \mathit{write} \wedge v.T_o \in \mathit{ancestor}(\mathcal{T})) \tag{29}$$

According to the post condition of Commit

$$(\mathit{root}(\mathcal{T}) \Rightarrow (v.S = \mathit{commit})) \wedge (\neg\mathit{root}(\mathcal{T}) \Rightarrow (v.\mathcal{T}_o = \mathit{parent}(\mathcal{T})) \tag{30}$$

ancestors may have locked the item when A-Write starts.

$$X.L.s = free \vee X.L.l = \{\mathcal{T}\} \vee \forall T_i \in X.L.l : T_i \in ancestor(\mathcal{T}) \tag{43}$$

Consequently, no new versions of an item X can be created with A-Write as long as leaf-transactions reading the same item have not yet committed. According to the postcondition of A-Commit:

$$\begin{aligned} root(\mathcal{T}) &\Rightarrow (X.L.l = \emptyset \wedge X.L.s = free) \\ \wedge\neg root(\mathcal{T}) &\Rightarrow (X.L.l = (L - \{\mathcal{T}\}) \cup \{parent(\mathcal{T})\}) \end{aligned} \tag{44}$$

The state of X is set to $free$ or the reading leaf-transaction identifier is replaced by its parent in the lock list. A new version of an item X can only be created by $s(e')$ when the transaction $d(e)$ that reads an earlier version has committed. At the local clock of X, $d(e)$ commits at time t before $s(e')$ writes, after which $s(e')$ commits at time $t' > t$. After the writing and commit by $s(e')$, the precondition of A-Read assures that no transactions can read the version written by $s(e)$.

$$\forall x \in X.V : v.\mathcal{T}_c \geq x.\mathcal{T}_c \tag{45}$$

Together with hypothesis 5: $d(e) \prec s(e')$.

OCC-BV Suppose that $d(e)$ is a W-transaction. The precondition of A-Validate and hypothesis 2 prevent that a new version is written and committed by $s(e')$ before the transaction $d(e)$ that reads an earlier version has committed.

$$\begin{aligned} T_L = (\mathbf{max}\ x \in X.V, (x.S = commit \vee x.\mathcal{T}_o \in ancestor(\mathcal{T})) : x.\mathcal{T}_c) \\ \wedge(\forall x \in X.V : x.\mathcal{T}_c \neq T_L \Rightarrow \mathcal{T} \notin x.list) \end{aligned} \tag{46}$$

According to the local clock of X, $s(e')$ writes after the validate/commit of $d(e)$. From hypothesis 5 follows: $s(e').\mathcal{T}_e > d(e).\mathcal{T}_e$. After the commit by $s(e')$ no transaction will read the version written by $s(e)$ as specified in the precondition of A-Read.

$$\forall x \in X.V - \{v\} : (x.S = commit \vee x.\mathcal{T}_o \in ancestor(\mathcal{T})) \Rightarrow v.\mathcal{T}_c > x.\mathcal{T}_c \tag{47}$$

Consequently, before and after the commit moment of $s(e')$ and according to the order definition: $d(e) \prec s(e')$.

OCC-BV & 2PL Suppose $d(e)$ is a RO-transaction. The RO-Read precondition

$$v.\mathcal{T}_c < \mathcal{T} \wedge v.S = commit \wedge (\forall x \in X.V - \{v\}, x.S = commit : x.\mathcal{T}_c < v.\mathcal{T}_c \vee x.\mathcal{T}_c > \mathcal{T}) \tag{48}$$

assures that the committed version of X with the largest timestamp $\mathcal{T}_c$ is chosen. Assume that this version was read at time t on the local clock of X. Suppose the transaction $s(e')$ commits at time $t' < t$ on the local clock of X. The precondition of RO-Read assures that $s(e).\mathcal{T}_e < d(e).\mathcal{T}_s < s(e').\mathcal{T}_e$. Suppose the transaction $s(e')$ commits at time $t' > t$ on the local clock of X. According to hypothesis 3, $s(e')$ commits with timestamp $s(e').\mathcal{T}_e > d(e).\mathcal{T}_s$. According to the order definition 2 for OCC-BV and 2PL: $d(e) \prec s(e')$.

the entry is changed to the parent of T_j. Define T_l, T_m as children of the owning transaction T_k with $T_j = T_l$ or T_l is an ancestor of T_j and $T_m = T_i$ or T_m is an ancestor of T_i. The pre- and post-conditions A-Validate

$$\text{PRE:} \qquad P(x) \equiv (\neg root(T_i) \Rightarrow x.T_o = parent(T_i)) \tag{40}$$

$$\text{POST:} \quad (\forall v \in X.V : v.T_o \neq T_i) \Rightarrow C_f = (\max x \in X.V, P(x) \wedge T_i \in x.list : x.C_b + 1) \tag{41}$$

together with the C_f calculation in the Validate PROCEDURE assure that $T_l.C_f > v.C_b$. The postcondition of A-Commit assures that the value of $v.C_b = T_m.C_f$. From the order definition 6, $T_m.C_f < T_l.C_f$, T_k is the parent of both T_m and T_l and $T_j \in ancestor(T_l) \vee T_j = T_l$ and $T_i \in ancestor(T_m) \vee T_i = T_m$ follows: $T_i \prec T_j$

Assume that T_j reads a committed version. The root of T_i has committed because the version has the state commit. T_j has not committed yet and has a different root: $T_i^* \neq T_j^*$. The same reasoning is valid as above. When the root of T_j commits the C_f value calculated for T_j^* in the Validate PROCEDURE is larger than the C_b value of v; $T_i^*.C_f = v.C_b > T_j^*.C_f$.

common From the order-definitions 6 and 5 it follows that the source of a labelled edge is ordered before its destination for all four CCA's. □

Theorem 5 (Leaf serializability) *The graph G' constructed from the ACTION specifications of section 8 is an acyclic TP(h).*

Proof. By construction, the vertex set of G is equal to the vertex set of G' and the edge set of G' is equal to the edge set of G with the addition of some unlabelled edges. An acyclic order is defined on all leaf-transactions T_i and for all $X \in D$ there is a directed path in G' that includes all X-edges. For two X-edges e and e' with different sources of which at least one is useful, their destinations are different, $d(e) \neq d(e')$, because a transaction reads a given value only from one transaction at the time. From lemma 11 it is known that $d(e) \succ s(e)$ and $d(e') \succ s(e')$.

Suppose that one of them, e' is useless. The case $s(e) \prec s(e')$ is proven in the same way as the case that both edges are useful. When $s(e') \prec s(e)$, $s(e)$ is either the direct successor of $s(e')$ or ordered after the direct successor of $s(e')$. The destination of the useless write, $d(e')$, is defined to be ordered before the direct successor of the source: $d(e') \prec s(e)$.

Suppose both e and e' are useful edges. In that case $s(e)$ and $s(e')$ are both W-transactions because no labelled edges exist from a RO-transaction or a from a vertex $T_{ix} \in H^u$. Without loss of generality it is assumed that $s(e) \prec s(e')$. When $d(e) = s(e')$, the theorem is trivially true by taking the X-path from T_0 to $s(e)$ and via $d(e) = s(e')$ to $d(e')$ and finally to T_f. When $d(e) \neq s(e')$, the case that $d(e)$ is a W-transaction is treated first, followed by the case that $d(e)$ is a RO-transaction.

The four CCA's are considered separately

2PL Suppose that $d(e)$ is a W-transaction. The postcondition of A-Read:

$$X.L.l = (L \cup \{T\}) \wedge (LS = free \Rightarrow X.L.s = read) \tag{42}$$

states that the lock-list, $X.L.l$, contains the identifier of the reading leaf-transaction and the state is unequal to *free*. The precondition of A-Write states that apart from itself only

MVTSO Suppose that $d(e)$ is a W-transaction. When $s(e')$ has already written a version before $d(e)$ reads, it follows for the precondition of A-Read that: $s(e') \succ d(e)$. When $s(e')$ writes a version after the reading by $d(e)$ the precondition in A-Write states:

$$\forall x \in X.V : (x.\mathcal{T}_w < \mathcal{T}_s \wedge x.\mathcal{T}_r < \mathcal{T}_s) \vee x.\mathcal{T}_w > \mathcal{T}_s \quad (49)$$

Consequently, after the reading by $d(e)$ only versions can be created by transaction $s(e')$ if $s(e').\mathcal{T}_s < s(e).\mathcal{T}_s \vee s(e').\mathcal{T}_s > d(e).\mathcal{T}_s > s(e).\mathcal{T}_s$. According to $s(e) \prec s(e')$ and the ordering definition: $d(e) \prec s(e')$.

Suppose $d(e)$ is a RO-transaction. The precondition of RO-Read

$$v.\mathcal{T}_w < \mathcal{T} \wedge v.S = commit \wedge (\forall x \in X.V : (x.\mathcal{T}_w \leq v.\mathcal{T}_w \vee x.\mathcal{T}_w > \mathcal{T}_s)) \quad (50)$$

assures that $s(e).\mathcal{T}_s < \mathcal{M}_{ro}$ and $s(e').\mathcal{T}_s > \mathcal{M}_{ro}$ when $s(e)$ and $s(e')$ are committed before $d(e)$ reads at time t on the local clock of X. The postcondition of RO-Read

$$Close(v) \quad (51)$$

assures that transactions writing a version, x, after the invocation of RO-Read cannot create this version with $x.\mathcal{T}_w < d(e).\mathcal{T}_s$. Consequently, the value of $\mathcal{M}_{ro}$ is not modified by transactions which commit after the execution of RO-read. When $s(e')$ commits at a time $t' > t$ on the local clock of X, hypothesis 3 states: $s(e').\mathcal{T}_e > d(e).\mathcal{T}_s$. Together with $s(e').\mathcal{T}_s > \mathcal{M}_{ro}$ it follows that: $d(e) \prec s(e')$.

OCC-NVTI The case RO-Read has been proven in theorem 2

Suppose that $d(e)$ is a W-transaction. From the assumption that $s(e) \prec s(e')$ it follows that $s(e).C_f < s(e').C_f$ and $v.C_b < v'.C_b$. Suppose $s(e')$ writes version v' which is an immediate successor of version v written by $s(e)$.

Assume that $s(e)$ and $s(e')$ have a common ancestor T_k, then $d(e)$ must have the same common ancestor. If this is not the case $d(e)$ has a root different from T_k or $d(e)$ and T_k share a common ancestor. In both cases $d(e)$ can only read the version written by $s(e)$ after the commit of the ancestor of T_k or its root. According to the precondition of A-Read, $d(e)$ would have read the version with the largest $\mathcal{T}_c$ value which is v'. This contradicts the assumption that $d(e)$ reads v. Consequently, $d(e)$ has the same ancestor T_k as $s(e)$ and $s(e')$.

Introduce three transactions $T_l, T_m, T_n \in child(T_k)$, with $T_l = s(e) \vee T_l \in ancestor(s(e))$ and $T_m = d(e) \vee T_m \in ancestor(d(e))$ and $T_n = s(e') \vee T_n \in ancestor(s(e'))$. After the reading of v by $d(e)$, an entry $d(e)$ is entered in $v.list$. When T_m commits before T_n the pre- and post-conditions of A-Commit:

$$\text{PRE:} \quad \forall v \in X.V : v.C_r = C_v \wedge v.list = L_v \wedge P(v) \equiv (X.\mathcal{T}_o = parent(T_i)) \quad (52)$$

$$\text{POST:} \quad \forall v \in X.V, T_i \in L_v : P(v) \Rightarrow v.C_r = \max(C_v, C_f) \quad (53)$$

assures that $v.C_r \geq T_m.C_f$. The postcondition of A-Validate and the calculation of the C_f-value in Validate assures that after invocation of A-Validate by T_n: $T_n.C_f > v.C_r \geq T_m.C_f$. From the order definition it follows that: $d(e) \prec s(e')$.

When T_n commits before T_m, $d(e)$ must already have read v, otherwise $d(e)$ would have read v'. Because v' is an immediate successor of v, the pre- and post-conditions of A-Commit:

$$\text{PRE:} \quad b \in X.V \wedge P(b) \wedge (\forall x \in X.V, P(x) : x.C_b \leq b.C_b) \quad (54)$$

$$\forall v \in X.V : v.\mathcal{T}_o = \mathcal{T}_{ov} \quad (55)$$

$$\text{POST:} \quad (\exists x \in X.V : \mathcal{T}_{ox} = T_i) \Rightarrow b.C_e = C_f \quad (56)$$

assure that $T_l.C_f = b.C_b < b.C_e \leq T_n.C_f$. The post-condition of A-Validate and the calculation of C_f assures that $T_m.C_f < T_m.C_l \leq b.C_e = T_n.C_f$. From the order definition it follows that $d(e) \prec s(e')$.

When $s(e)$ and $s(e')$ have no common ancestors, their roots are different. Exactly the same reasoning applies as for the case that both have the same ancestor.

When $s(e')$ is not an immediate sucessor to $s(e)$ an immediate successor $T_x \prec s(e')$ can be found. Above it is proven that $d(e) \prec T_x$. Combining both equations leads to $d(e) \prec s(e')$.

common From the ordering and the additional unlabelled edges added to G', an acyclic path can be constructed started at T_0 from $s(e)$ to $d(e)$, from $d(e)$ to $s(e')$ and from $s(e')$ to $d(e')$ to end at T_f. Consequently the graph G' constitutes a acyclic TP(h). □

It remains to prove that the intermediate transactions and the root transactions constitute a serializable history. In the graph G, the set N of vertices representing transactions which are children of a given transaction can be collapsed to form a single vertex. The vertex set N is replaced with one vertex T_n^1. All edges with both their source and their destination in N are removed. All edges with their source in N and their destination outside N have their source replaced by T_n^1; all edges with their destination in N and their sources outside N have their destination replaced with T_n^1. The thus created graph is called $G^{(1)}$. All edges in the graph $G^{(1)}$ with the same source, destination and label are replaced by one single edge with an identical label. In the graph $G^{(1)}$, unlabelled edges are drawn which connect the nested transactions and the leaf-transaction as specified for the construction of G' from G. This new graph is called $G^{(1)\prime}$. It then needs to be proven that this graph also represents a TP(h). This is a called L-serializability in [Vid91].

One of the requirements on the nested transactions are pointed out in section 4: when a nested transaction T_i precedes nested transaction T_j, then all descendants of T_i also precede T_j and all its descendants. The order number n of T_n^1 is chosen such that $n = (\mathbf{max}\ i : T_i \in child(T_n^1) : i)$. This order is reflected by the value of the timestamps attributed by the OCC and MVTSO algorithms.

Definition 7 (nested timestamps) *The start timestamp T_s^x of an intermediate/root transaction at level x is defined to be the minimum of the timestamps of all leaf-transactions which are descendants of this transaction. The end timestamp T_e^x of an intermediate/root transaction at level x is defined to be the maximum of the timestamps of all leaf-transactions which are descendants of this transaction.*

$$\begin{array}{l} T_s^x = (\mathbf{min}\ T \in descendant(T_n^x) : T.T_s) \\ T_e^x = (\mathbf{max}\ T \in descendant(T_n^x) : T.T_e) \end{array} \qquad (57)$$

The transactions T_{10} and T_{11} of the fig. 11 are collapsed to T_{11}^1. Graph G' shown in fig. 6 is modified to graph $G^{(1)\prime}$ shown in fig. 19. For the remainder of this paper: $T_i^0 \equiv T_i$. The following theorem is used:

Theorem 6 (L-serializability) *[Vid91] A history h is [L]-serializable iff both $G^{(1)\prime}$ and the history of the collapsed transaction are an acyclic TP(h).*

It is easily demonstrated that the history of the collapsed transaction is serializable. Remove in G all vertices and edges which have neither source nor destination in the collapsed

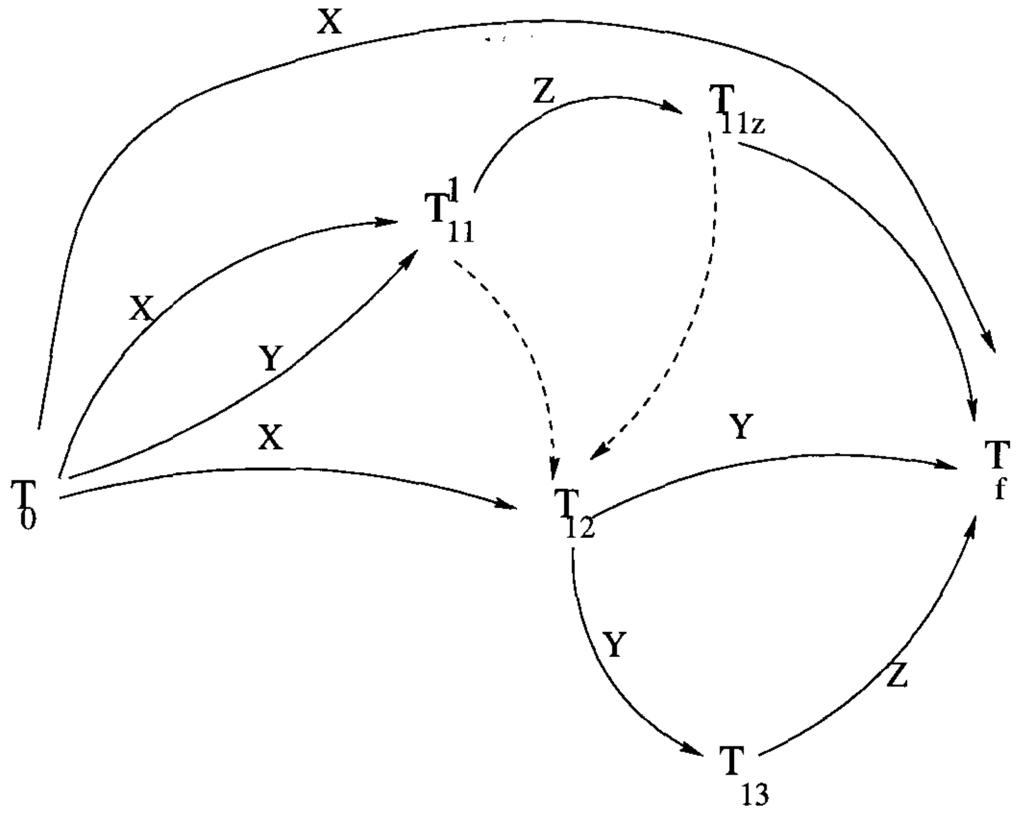

Figure 19: Example of graph $G^{(1)\prime}$ with collapsed T_{10} and T_{11}

transaction. Replace the source of the ingoing edges by T_0 and the destinations of the outgoing edges by T_f. View serializability (theorem 5) has already been proven for all edges with both vertices representing leaf-transactions of the collapsed transaction. For the other edges, the lemma is trivially true from the definition of the ordering of T_0 and T_f. The serializability of the history of the collapsed transaction follows. Below it is proven that $G^{(1)\prime}$ is an acyclic TP(h).

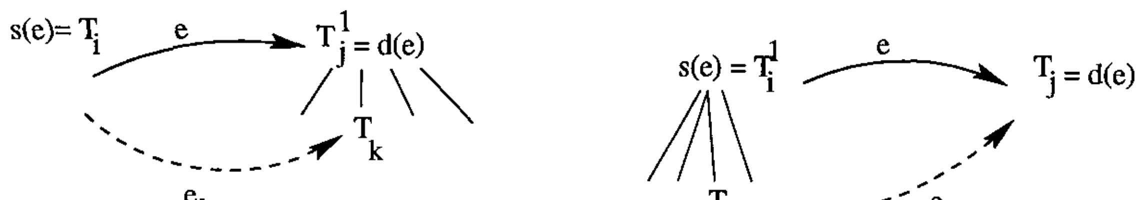

Figure 20: Collapsing of either $s(e)$ or $d(e)$

Lemma 12 *For any labelled edge e from $G^{(1)}$, with $s(e) = T_i^x$ and $d(e) = T_j^y$, where one is a collapsed parent transaction and the other a leaf-transaction not a descendant of the first $((x = 0 \wedge y = 1) \vee (x = 1 \wedge y = 0))$, it holds that $T_i^x \prec T_j^y$.*

Proof. Consider the first alternative that $T_j^1 = d(e)$ is a collapsed parent with children T_k. For every labelled edge e_k with $s(e_k) = T_i$ and $d(e_k) = T_k$, lemmas 4 and 11 state that

$T_i \prec T_k$. All these edges, e_k with $s(e_k) = T_i$ and $d(e_k) = T_k$, are replaced by the edges e with $s(e) = T_i$ and $d(e) = T_j^1$. Edges with the same label and source are collapsed to one edge (see fig. 20). It is then required that for all e_k with the label X, every X-writing $s(e)$ is ordered before all X-reading T_k.

2PL The collapsed transaction can belong to a different root from T_i or to the same one. If the roots are different, the version of T_i can only be read when the root has committed. The reading transactions, T_k, commits after the commit of T_i and from hypothesis 5 it follows that the above condition is satisfied. When the collapsed transaction belongs to the same root, T_i is not a descendant of T_j^1. All T_k can only start reading when the version written by T_i is owned by an ancestor of T_k. This has as consequence that an ancestor of T_i has committed. T_k can only commit after this ancestor and this one in turn can only commit after T_i. Consequently, T_i is ordered before all T_k.

OCC-BV Similar to 2PL proof.

MVTSO $T_j^1 \notin \mathit{descendant}(T_i)$ and $T_i.\mathcal{T}_s < T_j^1.\mathcal{T}_s = \mathit{parent}(T_k).\mathcal{T}_s$ implies according to hypothesis 6 that: $T_i \prec T_k$.

OCC-NVTI Suppose that $s(e)$ and $d(e)$ have different roots $s(e)^*$ and $d(e)^*$. According to definition 6: $s(e)^*.C_f < d(e)^*.C_f$. All children T_k of $d(e)$ have the same root. According to the order definition: $\forall T_k : s(e) \prec T_k$. When $s(e)$ and $d(e)$ have the same root, they have a common ancestor with two children T_m and T_l with $T_m.C_f < T_l.C_f$ and $s(e) = T_m \vee s(e) \in \mathit{descendant}(T_m)$ and $d(e) = T_l \vee d(e) \in \mathit{descendant}(T_l)$. The same reasoning as developed for $s(e)^*$ and $d(e)^*$ apllies to T_m and T_l, which leads to the same conclusion: $\forall T_k : s(e) \prec T_k$.

common Consequently, the lemma is satisfied for all four CCA's when the destination is the collapsed transaction.

Consider the second alternative that T_i^1 = s(e) is a collapsed parent with children T_k. For every labelled edge e_k with $d(e_k) = T_j$ and $s(e_k) = T_k$, lemma 4 states that $T_k \prec T_j$. Again, edges with the same label X and the same destination $d(e)$ are collapsed. Consequently, for all e_k with the same label X, all T_k must be ordered before every $d(e)$.

2PL When T_i^1 and T_j have different roots, all leaf transactions T_k must have committed before T_j can read them. T_j commits after T_i^1 and (hypothesis 5) each T_j is ordered after all T_k.

Suppose the collapsed transaction and the reading transaction T_j have the same root. The version wriiten by one of the descendants of $s(e)$ has the state *write* when it is read by $d(e)$. According to the precondition of A-Read:

$$X.L.s = \mathit{write} \wedge v.T_o \in \mathit{ancestor}(\mathcal{T}) \tag{58}$$

the leaf-transaction T_j can only read the versions of the children of T_i^1 when an ancestor of T_j is owner of these versions. This happens after the commit of T_i^1 which only occurs after the last commit of all children T_k of T_i^1. Consequently, T_j can only commit after the last commit of T_k.

OCC-BV When T_i^1 and T_j have different roots, all leaf transactions T_k must have committed before T_j can read them. T_j commits after T_i^1 and (hypothesis 5) each T_j is ordered after all T_k.

Suppose the collapsed transaction and the reading transaction T_j have the same root. A leaf-transaction can only read a version that is committed or a tentative version that has a common ancestor. The latter implies that T_i^1 has committed and consequently all its children T_k have committed before the versions can be read. The reading transactions T_j commits after T_i^1 and it follows that $T_k \prec T_j$ for all k.

MVTSO $parent(T_k).\mathcal{T}_s = T_i^1.\mathcal{T}_s < T_j.\mathcal{T}_s$ and $T_j \notin descendant(T_i^1)$ implies according to hypothesis 6 that: $T_k \prec T_j$.

OCC-NVTI Identical to the OCC-NVTI proof for the collapsed $s(e)$. □

Lemma 13 (Collapsed serializability) *The graph $G^{(1)\prime}$ constructed from $G^{(1)}$ where a set of leaf-transactions is replaced by their parent, is an acyclic TP(h).*

Proof. The part of the proof that concerns RO-transactions and useless writes, is identical to the proof of theorem 2. Consider two X-edges e and e' where both sources and destinations are W-transactions. When all edges are simple leaf-transactions, theorem 2 applies. The cases that one of the four vertices are collapsed parents is of interest. Without loss of generality take $s(e) \prec s(e')$. When $s(e)$ or $d(e')$ is a collapsed parent the theorem is trivially true by applying theorem 2 and lemma 12. Two cases are left: (1) $s(e')$ is a collapsed parent and $d(e)$ is a leaf-transaction and (2) $d(e)$ is a collapsed parent and $s(e')$ is a leaf-transaction.

Ad 1) When $d(e)$ is a child of $s(e')$, $d(e) = s(e')$ after the collapse of $d(e)$. Applying the lemma 12 twice proves the lemma. Take the case that $d(e)$ is not a child of $s(e')$. It must be proven that $d(e) \prec s(e')$. This is equivalent with proving that all children T_k of $s(e')$ obey $d(e) \prec T_k$ for every $d(e)$.

2PL $d(e)$ reads before any descendant of $s(e')$ commits, otherwise $d(e)$ would have read a version written by a descendant of $s(e')$. A child of $s(e')$ can only write a version when the lock is held by its ancestors or is free. This means that either $d(e)$ has committed and the lock is passed to a common ancestor of $d(e)$ and $s(e')$, or that the lock is released and $d(e)$ is the descendant of a root different from the root of $s(e')$. In both cases $d(e)$ has committed before all writing children of $s(e')$.

OCC-BV The precondition on A-Commit

$$\forall x \in X.V - C : (x.\mathcal{T}_c < (\mathbf{min}\ v \in C : v.\mathcal{T}_c)) \vee (x.\mathcal{T}_c > (\mathbf{max}\ v \in C : v.\mathcal{T}_c)) \tag{59}$$

assures that all commit by children of $s(e')$ occur either before or after the commit of $s(e)$. From the assumption that $s(e) \prec s(e')$ it follows that all children T_k of $s(e')$ commit after $s(e)$. The pre-condition of A-Validate

$$\begin{gathered} T_L = (\mathbf{max}\ x \in X.V, (x.S = commit \vee x.\mathcal{T}_o \in ancestor(\mathcal{T})) : x.\mathcal{T}_c) \\ \wedge(\forall x \in X.V : x.\mathcal{T}_c \neq T_L \Rightarrow \mathcal{T} \notin x.list) \end{gathered} \tag{60}$$

assures that all children of $s(e')$ commit after the the commit of $d(e)$. From hypothesis 5 it follows that:

$$\forall T_k, T_k \in child(s(e')) : d(e).\mathcal{T}_e < T_k.\mathcal{T}_e \tag{61}$$

Consequently, $s(e) \prec d(e) \prec s(e')$.

MVTSO The application of hypothesis 6 validates the lemma.

OCC-NVTI The same reasoning as used in lemma 12 applies.

Ad 2) $d(e)$ is a collapsed parent.

2PL The precondition of A-Write states that all locking transactions are ancestors of $s(e')$.

$$\forall T_i \in X.L.l : T_i \in ancestor(\mathcal{T}) \tag{62}$$

Consequently, all leaf-transactions T_k of $d(e)$ must have committed before the write can start. Because $d(e)$ commits later than the committed leaf-transactions, it follows that for all T_k, $T_k \prec s(e')$.

OCC-BV The precondition on A-Validate

$$\begin{array}{c} T_L = (\mathbf{max}\, x \in X.V, (x.S = commit \vee x.\mathcal{T}_o \in ancestor(\mathcal{T})) : x.\mathcal{T}_c) \\ \wedge(\forall x \in X.V : x.\mathcal{T}_c \neq T_L \Rightarrow \mathcal{T} \notin x.list) \end{array} \tag{63}$$

assures that no new version can be added between the commit of the reading transactions and the commit of the writing transaction $s(e)$. Consequently, all children of $d(e)$ have committed before the new version of $s(e')$ can be installed. Together with hypothesis 5, this implies that $s(e') \succ d(e)$.

MVTSO Hypothesis 6 assures that $d(e) \prec s(e')$.

OCC-NVTI The same reasoning as applied in lemma 12 applies.

common In all cases it is true that $s(e) \prec d(e)$ and $s(e') \prec d(e')$ and $d(e) \prec s(e') \vee d(e) = s(e')$, which proves the lemma. □

Transactions can be collapsed to their parents and in their turn these parents can be collapsed to their parents. Root-transactions cannot further be collapsed. A completely collapsed history is the history of the root-transactions. The set which consists of history h of leaf-transactions and all collapsed histories derived from h is called the *collapsed history set*, $CH(h)$.

Theorem 7 *[Nested Serializability] All collapsed histories $ch \in CH(h)$ with h a history governed by the ACTION specifications of section 4 are serializable histories.*

Proof. For theorem 5 it is proven that h is a serializable history. In lemma 13, it is proven that a singly collapsed history $h^{(1)}$ is a serializable history because the derived graph $G^{(1)\prime}$ is a acyclic TP(h). Remove in G for all $X \in D$ all write and read actions on X where W[X] precedes R[X] in the collapsed parent and replace them with equivalent write and read actions on internal variables of the collapsed parent. The modified collapsed parent has now all the characteristics of a leaf-transaction. $G^{(1)}$ can be considered a graph of leaf-transactions.

Consider a given graph $G^{(k+1)}$ recursively constructed by collapsing a parent of the graph constructed in $G^{(k)}$. Assume that $G^{(k)}$ consists of leaf-transactions and that a graph $G^{(k)\prime}$ can be constructed which is a TP(h). Collapsing a parent in $G^{(k)}$ creates the graph $G^{(k+1)}$. The graph $G^{(k+1)\prime}$ can then be proven to be a TP(h) as shown by lemma 13. In the graph $G^{(k+1)}$, the single collapsed transaction can be replaced by an equivalent leaf-transaction as shown for $G^{(1)}$. □

Theorem 8 (Exception atomicity) *The versions created by a transaction are visible to all following transactions or to none.*

Proof. A version's state can be committed or tentative. When the version's state is committed, the version cannot be removed and is consequently visible to all transactions following the committing transaction. According to all A-Read preconditions

$$v.S = commit \vee v.T_o \in ancestor(T) \tag{64}$$

versions with the state *tentative* are only visible to transactions which are a descendant of the owning transaction. Other transactions see the version only after its commit. According to the A-Abort pre- and post-conditions,

$$\begin{array}{ll} \text{PRE:} & X.V = Q \wedge C = \{v \mid v \in X.V \wedge v.T_o = T\} \\ \text{POST:} & X.V = Q - C \end{array} \tag{65}$$

versions are removed when the owning transaction is aborted. The owning transaction owns the versions written by all descendants as specified by the postcondition of A-Commit:

$$\neg root(T) \Rightarrow v.T_o = parent(T) \tag{66}$$

When the owning transaction aborts, the results of all its descendants are removed as well. The removed version was only read by aborted descendants of the owning transaction. Consequently, no transaction has had access to the version under consideration. □

9.1 Proof of HRT/SRT serializability

The hrt-order is already defined for OCC-VTI in section 7. The hrt order can be defined for 2PL, OCC-BV and MVTSO:

Definition 8 (hrt-order extension) *For any two transactions T_i and $T_j, T_i \overset{X}{\prec} T_j$ iff*

- *T_i and T_j are leaf-transactions of the same class and ordered according to definitions 5 and 2.*

- *T_i writes $X \in HO \cup SO$ and $T_j = T_{id}$.*
- *T_j accesses $X \in HO \cup SO$ and T_i is a specification transaction and $T_i.\mathcal{T}_y < T_j.\mathcal{T}_y$, with $\mathcal{T}_y = \mathcal{T}_s$ for MVTSO and $\mathcal{T}_y = \mathcal{T}_e$ for 2PL & OCC-BV.*
- *T_i accesses $X \in HO \cup SO$ and T_j is a specification transaction and $T_i.\mathcal{T}_y < T_j.\mathcal{T}_y$, with $\mathcal{T}_y = \mathcal{T}_s$ for MVTSO and $\mathcal{T}_y = \mathcal{T}_e$ for 2PL & OCC-BV.*

It is important to note that the transfer-serializability concept is independent of the choice of CCA. The SRT-transactions may use a different CCA from the HRT-transacions.

Lemma 14 *The order of definition 8 is acyclic.*

Proof. Consider the ordering without the specification transactions. Two mutually unordered sets of HRT and SRT transactions exist. According to theorem 7 each individual set is ordered. The addition of the specification transactions can lead to cycles in two ways: (1) when a pair of HRT(SRT)-transactions are differently ordered with respect to each other than with respect to a given specification transaction and (2) when a cyclic ordering can be found from a specification transaction T_{id} via a set of HRT(SRT) transactions to another specification transaction $T_{jd} \neq T_{id}$ and from T_{jd} via a set of SRT(HRT) transactions back to T_{id}.

ad 1) Consider that the following order is valid: $T_j \overset{X'}{\prec} T_{id} \overset{Y'}{\prec} T_k$, with T_j and T_k HRT(SRT)-transactions of the same class. In case of MVTSO, $T_j.\mathcal{T}_s < T_{id}.\mathcal{T}_s < T_k.\mathcal{T}_s$ implies that for conflicting T_j and T_k: $T_j \prec T_k$. In case of OCC-BV and 2PL, $T_j.\mathcal{T}_e \leq T_{id}.\mathcal{T}_e < T_k.\mathcal{T}_e$ implies that for conflicting T_j and T_k: $T_j \prec T_k$. Consequently, no cycle is possible.

ad 2) Without loss of generality, assume that a chain of HRT(SRT)-transactions is ordered after the specification transaction T_{id} and before the specification transactions T_{jd}. According to the definition 4: $T_{id}.\mathcal{T}_y < T_{jd}.\mathcal{T}_y$ with $\mathcal{T}_y = \mathcal{T}_s$ for MVTSO and $\mathcal{T}_y = \mathcal{T}_e$ for OCC-BV and 2PL. To create a cycle, a chain of SRT(HRT)-transactions is ordered before the specification transaction T_{id} and after the specification transactions T_{jd}. According to the definition 4: $T_{id}.\mathcal{T}_y > T_{jd}.\mathcal{T}_y$ with $\mathcal{T}_y = \mathcal{T}_s$ for MVTSO and $\mathcal{T}_y = \mathcal{T}_e$ for OCC-BV and 2PL. This constitutes a contradiction. Consequently, the order is acyclic. □

It is proven that the graph G'_{hrt} represents a *transfer-serializable* history for the leaf-transactions which constitute h.

Lemma 15 *For any Y-labelled edge e from G_{hrt}, with $Y = X$ or $Y = X'$, $s(e)$ and $d(e)$ both leaf-transactions, it holds that $s(e) \overset{Y}{\prec} d(e)$.*

Proof. When s(e) and d(e) are both HRT- or SRT-transactions, lemma 12 applies and $s(e) \overset{X}{\prec} d(e)$.

One of the two can be a specification transaction. When d(e) is a specification transaction, $s(e) \prec d(e)$ is determined by the hrt-order extension definition. When s(e) is a specification transaction, there exists a T_i such that $s(e) = T_{id}$. When T_i is a HRT(SRT)-transactions then $d(e)$ is a SRT(HRT) transactions. Consequently, they cannot have a common ancestor.

In case of 2PL and OCC-BV, $d(e)$ only reads X' after the commit of T_i at local time t. According to the precondition of TR-Read, the commit of $d(e)$ at local time $t' > t$ and hypothesis 5: $d(e).\mathcal{T}_e > T_i.\mathcal{T}_e$. According to the definition of specification transaction timestamps: $s(e).\mathcal{T}_e = T_i.\mathcal{T}_e$. Combination of both conditions leads to the conclusion that $s(e) \overset{X'}{\prec} d(e)$.

In case of MVTSO, $d(e)$ reads X' after the commit of T_i. From the precondition of TR-Read follows: $d(e).\mathcal{T}_s > T_i.\mathcal{T}_e$ and from the definition of the specification transaction follows: $s(e).\mathcal{T}_s = T_i.\mathcal{T}_e$. Their combination leads to the conclusion that $s(e) \overset{X'}{\prec} d(e)$. Consequently, combining all cases, it follows that $s(e) \overset{Y}{\prec} d(e)$. □

Theorem 9 (Transfer-serializability) *The graph G'_{hrt} constructed from the ACTION specifications of section 4 and transfer-edges is an acyclic TP(h).*

Proof. For two X-edges e and e' with different sources of which at least one is useful, say e', their destinations are different, $d(e) \neq d(e')$, because a transaction reads a given value only from one transaction at the time. Without loss of generality it is assumed that $s(e) \prec s(e')$. From lemma 7 it is known that $d(e) \succ s(e)$ and $d(e') \succ s(e')$. By construction and by the definition of the transfer-edges, HRT transactions access other data-items than SRT-transactions and are not related. The identical labels of e and e' imply that when one of the four transactions is a HRT(SRT) transaction then the other transactions are either HRT(SRT) transactions or specification transactions.

When $s(e)$, $d(e)$, $s(e')$ and $d(e')$ are all HRT-transactions or SRT-transactions, theorem 7 applies.

When $s(e)$ is a specification transaction, $s(e')$ is necessarily also a specification transaction because both edges, e and e' are transfer-edges labeled with the same label X' and only specification transactions are the sources of transfer-edges. Consequently, only two cases need to be considered: (1) $d(e)$ is a specification transaction and (2) $s(e')$ is a specification transaction.

Ad (1): According to the definition of transfer transactions, $d(e)$ is ordered before the immediate successor of $s(e)$. $s(e')$ belongs to the same class as $s(e)$ (HRT or SRT) and is the immediate successor of $s(e)$ or $s(e')$ is ordered after the immediate successor and therefore: $d(e) \prec s(e')$.

Ad (2): There exists a transaction T_i corresponding with the specification transaction $T_{id} = s(e)$ and a transaction T_j corresponding with transaction $T_{jd} = s(e')$. The SRT(HRT)-transaction $d(e)$ has a different root from the HRT(SRT)-transactions T_i and T_j. T_i has a different root from T_j as $d(e)$ considers them as two separate transactions.

In case of OCC-BV and 2PL, the ordering $s(e) \prec s(e')$ implies that $T_i.\mathcal{T}_e < T_j.\mathcal{T}_e$. Assume that TR-Read is invoked at local time t. T_i has committed before TR-Read is invoked, otherwise $d(e)$ could not have read the value written by T_i. Two cases must be discerned: (1) T_j commits before t and (2) T_j commits after t. In case 1 the precondition of TR-Read stipulates that $T_i.\mathcal{T}_e < d(e).\mathcal{T}_s < T_j.\mathcal{T}_e$. In case 2 hypothesis 4 assures also: $T_i.\mathcal{T}_e < d(e).\mathcal{T}_s < T_j.\mathcal{T}_e$. Using the definition of specification transaction timestamps, the substitution $s(e').\mathcal{T}_e = T_j.\mathcal{T}_e$ yields: $d(e).\mathcal{T}_s < s(e').\mathcal{T}_e$. Which leads to: $s(e) \overset{X'}{\prec} d(e) \overset{X'}{\prec} s(e') \overset{X'}{\prec} d(e')$.

In case of MVTSO, The precondition of TR-Read implies: $s(e).\mathcal{T}_s = T_i.\mathcal{T}_e < d(e).\mathcal{T}_s < T_j.\mathcal{T}_e$ when T_j commits before $d(e)$ reads. Hypothesis 4 assures that the same is true when T_j commits after the read-action of $d(e)$. Further, the definition of the specification transaction timestamp implies: $s(e').\mathcal{T}_s = T_j.\mathcal{T}_e$. Combining the two equations above yields: $s(e).\mathcal{T}_s < d(e).\mathcal{T}_s < s(e').\mathcal{T}_s$.

A path can be constructed from T_0 via the total order to $s(e)$, from $s(e)$ via $d(e)$ and $s(e')$ to $d(e)$ and from $d(e)$ via the acyclic order to T_f. Which concludes the proof of the theorem. □

10 Implementation

10.1 Timestamp creation

During run-time, transactions are identified with a timestamp.

```
PROCEDURE TimeStamp( T: Timestamp): TimeStamp
BEGIN
        IF T =0 THEN TimeStamp := time ∘ PID
        ELSE TimeStamp := T ∘ OID
END
```

Figure 21: Creation of Timestamps

In Fig. 21, the creation of a timestamp is shown. A root-transaction invokes TimeStamp with $T = 0$ as parameter. The timestamp is returned as the local time suffixed with the Processor IDentifier (PID). This establishes unique timestamps. Hypothesis 6 requires that when root transaction T_1 precedes T_2, all descendants of T_1 precede T_2 and precede all descendants of T_2. When the timestamp is generated for an intermediate transaction, the timestamp of the parent-transaction suffixed with the Object identifier OID is returned. It is assumed that (possibly in time) unique OID can be generated. These timestamps allow to check if a given timestamp $\mathcal{T}_i$ is the ancestor of another transaction represented with timestamp $\mathcal{T}_j$. This relation is recursively represented with:

$$T_i \in ancestor(T_j) \Leftrightarrow \begin{cases} \mathcal{T}_j = \mathcal{T}_i \circ OID \\ \mathcal{T}_j = \mathcal{T}_k \circ OID \wedge T_k \in ancestor(T_i) \end{cases} \tag{67}$$

A relation is defined for any two timestamps $\mathcal{T}_i$ and $\mathcal{T}_j$ where K_i and K_j are integer values:

$$\mathcal{T}_i < \mathcal{T}_j \Leftrightarrow \begin{cases} \mathcal{T}_j = \mathcal{T}_i \circ \mathcal{T}' & (a) \\ \mathcal{T}_i = K \circ \mathcal{T}'_i \wedge \mathcal{T}_j = K \circ \mathcal{T}'_j \wedge \mathcal{T}'_i < \mathcal{T}'_j & (b) \\ \mathcal{T}_i = K_i \circ \mathcal{T} \wedge \mathcal{T}_j = K_j \circ \mathcal{T}' \wedge K_i < K_j & (c) \end{cases} \tag{68}$$

The above relation for the timestamps satisfies the hypothesis 6 for MVTSO discussed in section 6.

Lemma 16 *($T_i.\mathcal{T}_s < parent(T_k).\mathcal{T}_s < T_j.\mathcal{T}_s \wedge T_j \notin descendant(parent(T_k)) \wedge parent(T_k) \notin descendant(T_i)) \Rightarrow T_i.\mathcal{T}_s < T_k.\mathcal{T}_s < T_j.\mathcal{T}_s$*

Proof. From $T_i.\mathcal{T}_s < parent(T_k).\mathcal{T}_s$ and equations 67 and 68 follows that

$$T_i \notin descendant(parent(T_k)) \tag{69}$$

Similarly it follows that $parent(T_k) \notin descendant(T_j)$. The line (a) of equation 68 which defines the order for descendants does not apply:

$$\begin{array}{l} T_j.\mathcal{T}_s \neq parent(T_k).\mathcal{T}_s \circ \mathcal{T} \\ parent(T_k).\mathcal{T}_s \neq T_i.\mathcal{T}_s \circ \mathcal{T} \end{array} \tag{70}$$

The lines (b) and (c) of equation 68 apply. This means that the timstamps of T_i and $parent(T_k)$ consist of an equal number ($n > 0$) of identical integers followed by integer values

K_i and K_k, with $K_i < K_k$, followed by an undetermined number of integer values. The same applies to $parent(T_k)$ and T_j.

$$\begin{array}{l} parent(T_k).\mathcal{T}_s = \mathcal{T} \circ K_k \circ \mathcal{T}_k < \mathcal{T} \circ K_j \circ \mathcal{T}_j = T_j.\mathcal{T}_s \\ T_i.\mathcal{T}_s = \mathcal{T}' \circ K_i \circ \mathcal{T}_i < \mathcal{T}' \circ K'_k \circ \mathcal{T}'_k = parent(T_k).\mathcal{T}_s \end{array} \tag{71}$$

where $\mathcal{T}$, $\mathcal{T}'$, $\mathcal{T}_i$, $\mathcal{T}_j$, $\mathcal{T}'_k$ and $\mathcal{T}_k$ are possibly empty sequences of integers and K_i and K_j are integer values. According to the construction of the timestamp shown in fig. 21:

$$(parent(T_k).\mathcal{T}_s) \circ A_k = T_k.\mathcal{T}_s \tag{72}$$

Inserting in equation 71 the result from eq. :68, $\mathcal{T}_i < \mathcal{T}_i \circ A_k$ yields:

$$\begin{array}{l} parent(T_k).\mathcal{T}_s = \mathcal{T} \circ K_k \circ \mathcal{T}_k < \mathcal{T} \circ K_k \circ \mathcal{T}_k \circ A_k < \mathcal{T} \circ K_j \circ \mathcal{T}_j = T_j.\mathcal{T}_s \\ T_i.\mathcal{T}_s = \mathcal{T}' \circ K_i \circ \mathcal{T}_i < \mathcal{T}' \circ K'_k \circ \mathcal{T}'_k < \mathcal{T}' \circ K'_k \circ \mathcal{T}'_k \circ A_k = T_k.\mathcal{T}_s \end{array} \tag{73}$$

Substituting equations 72 and 71 the final result is obtained:

$$\begin{array}{l} parent(T_k).\mathcal{T}_s < T_k.\mathcal{T}_s < T_j.\mathcal{T}_s \\ T_i.\mathcal{T}_s < parent(T_k).\mathcal{T}_s < T_k.\mathcal{T}_s \end{array} \tag{74}$$

Their combination and removal of $parent(T_k).\mathcal{T}_s$ leads to the wanted result. $\square$

10.2 System model

In this section the underlying assumptions and services are discussed. It is assumed that the underlying system is composed of processors interconnected via links. Processors are fail-silent: they behave according to specification until the moment of failure after which they perform no detectable actions. The following services are available.

Multicast service delivers messages to a set of processes with the following characteristics:

- **Integrity** A message received by a receiver is sent by a sender.
- **Order** Two HRT commit messages m and m' sent at local times t_s and t'_s, with $t_s < t'_s$, will be received at local times $t_r < t'_r$.
- **Validity** Messages from a correct sender are received by all correct receivers.
- **Unanimity** A message is received by all correct receivers or by none.
- **Timeliness** A message sent by a HRT sender arrives within bounded time at all correct HRT receivers.

Clock synchronization service assures that correct processors have local clocks which are approximately synchronized with a known maximum deviation ϵ. Clock values read at different instants are monotonically increasing. $C_p(t)$ denotes for each processor p its clock value at real time t. For any two processors p and q, it holds that $\mid C_p(t) - C_q(t) \mid < \epsilon$.

Online HRT scheduler maintains a list of executions, processors, start-times and deadlines, that constitutes the HRT schedule. The online scheduler starts and stops executions to assure that the executions obey the online schedule. Processors are replicated and a replicated group is assumed not to fail such that all scheduled actions can be succesfully completed.

Communication delay The communication duration Δ of a multicast and the communication duration δ of a point to point message is related to the clock deviations. When a message is sent (multicast) by sender, p, at local clock time $C_p(t)$ and received by receiver, q, at local clock time $C_q(t')$ then the following relation holds: $0 < C_q(t') - C_p(t) \leq \Delta(\delta)$.

Hypothesis 5 can be restated as lemma 17.

Lemma 17 *In case of 2PL, OCC-NVTI and OCC-BV, two RW-conflicting transactions T and T', with commit-timestamps $\mathcal{T}_e$ and $\mathcal{T'}_e$ execute their commit-actions at local times $C_q(t)$ and $C_q(t')$ such that: $C_q(t) > C_q(t') \Leftrightarrow \mathcal{T}_e > \mathcal{T'}_e$.*

Proof. Suppose two RW-conflicting transaction T and T' exist.

2PL Without loss of generality, T' has locked the conflicting item on processor q before T. The precondition of T is only fulfilled after the liberation of the data-item by T'. The execution by T' of A-Commit on processor q at local time $C_q(t')$ liberates the item. The commits of T and T' will be initiated by the coordinator c' at a local time $C_{c'}(t'_c)$ and at coordinator c at a local time $C_c(t_c)$. Due to the clock synchronization and the communication delay, c' sent this message at local time $C_{c'}(t'_s) \geq C_q(t')$. The value of the data-item will be read by T at a local time $C_q(t_s) > C_q(t')$. Due to the clock synchronization and the communication delay, the message will be received by c at a local time $C_c(t_r) \geq C_q(t_s)$. This means that

$$C_q(t) > C_q(t') \Rightarrow C_{c'}(t'_c) < C_c(t_c). \tag{75}$$

Inversely, when $C_{c'}(t'_c) < C_c(t_c)$ the liberation of the item occurs before the the start of the commit: $C_{c'}(t'_c) > C_q(t)$. Using $C_q(t) > C_c(t_c)$ leads to:

$$C_q(t) > C_q(t') \Leftarrow C_{c'}(t'_c) < C_c(t_c). \tag{76}$$

According to the construction of the timestamps at local times $C_{c'}(t'_c)$ and $C_c(t_c)$, the accompanying commit timestamps will obey: $C_{c'}(t'_c) < C_c(t_c) \Leftrightarrow TS(C_{c'}(t'_c)) < TS(C_c(t_c))$ and $TS(C_{c'}(t'_c)) < TS(C_c(t_c)) \Leftrightarrow \mathcal{T'}_e < \mathcal{T}_e$. Combining the two last equations with equations 75 and 76 yields the wanted result.

OCC-BV & OCC-NVTI The validate/commit action pair of two transactions that write to the same item X is done in a strictly serial order because only one transaction at the time can add a new version with state *validating* to the set of versions of X. The minimum difference between the timestamps, $\mathcal{T}_c$, $\mathcal{T}'_c$ of two consecutively created versions is determined by the maximum deviations between the local clocks and the minimum communication delay δ. The difference between local time t of the creation of the timestamp $\mathcal{T}_c$ and the local time t' of the creation of the next timestamp $\mathcal{T}'_c$ is determined by the sending of a message to the coordinator and the sending of the commit message by the coordinator: $t' - t > 2.\delta$. The chosen timestamp $\mathcal{T}_c$ is taken from the maxima of the returned timestamps such that $\mathcal{T}_c = timestamp(t_s) \wedge t_s - t < \epsilon < \delta$. For a minimum difference between $\mathcal{T}_c$ and $\mathcal{T}'_c$: $\mathcal{T}'_c = timestamp(t')$. This has as consequence that for any two consecutively created version with $\mathcal{T}_c = timestamp(t)$ and $\mathcal{T}'_c = timestamp(t')$ that $t' - t > \delta$

Two cases must be proven to be impossible: (1) a writing transaction commits after the commit of a reading transaction with a larger commit-timestamp and (2) a reading transaction commits after the commit of a writing transaction with a larger commit-timestamp.

Ad 1) The timestamps of the reading transaction $\mathcal{T}_r$ and the writing transaction $\mathcal{T}_w$, with $\mathcal{T}_w > \mathcal{T}_r$, are generated from local times t_r and t_w respectively. In this case the reading transaction has read an earlier version with timestamp $\mathcal{T}_c$ generated at local time t_c. Due to the precondition of A-Validate the writing transaction with the larger commit timestamp invokes A-Validate after the invocation of A-Validate by the reading transaction. In this case $t_r - t_c < \delta$ and $t_w - t_c > \delta$ as stated by the lemma.

Ad 2) A reading transaction that commits with timestamp $\mathcal{T}_r$ after another writing transaction with commit timestamp $\mathcal{T}_w$ has either read the version written by this transaction or has invoked A-Validate before the writing transaction. In the first case the creation of the commit timestamp assures by construction that $\mathcal{T}_r > \mathcal{T}_w$. The second case is similar to the one treated in ad 1). □

10.3 Commit algorithms

The correctness of the specified transactions strongly depends on a timely commit implementation. During a commit all versions created by the transaction under consideration are either installed and committed or removed. It is important that all related versions reach the same state after the commit. In addition, the commit should be executed in bounded time. The here presented commit algorithms depend on the above presented services. It is assumed that there is one coordinator process which initiates the transaction and at the end of the transaction also initiates the commit. By construction only one coordinator exists and when no faults occur, a coordinator will be present. When needed, the coordinator waits a fixed time on answers from the participants in the commit protocol. When this time has elapsed without response from some participant, the coordinator will assume that a failure occurred. Participants will also wait for the coordinator's response on their answer to the coordinator. When this time has elapsed, participants will assume a failure occurred and will send an "abort" message to all participants. This way the non-blocking property of the commit protocol is acquired.

The requirements on the Commit are:

Unanimity All participants that decide reach the same decision.

Integrity Only when all (correct) participants agree to commit, can a participant commit.

Validity When all participants agree to commit and no failures occur, then all participants commit.

Ordering Two transactions T and T' that execute A-Commit at local times $C_p(t)$ and $C_p(t')$ with commit timestamps, $\mathcal{T}_e$ and $\mathcal{T}'_e$ such that $C_p(t) < C_p(t') \Leftrightarrow \mathcal{T}_e < \mathcal{T}'_e$.

Timeliness When the coordinator starts the commit at time $C_c(t)$ at its local clock, then there is a constant d such that all correct participants have either committed or aborted at a local time $C_p(t') = C_c(t) + d$.

In addition, hypotheses 2, 3 and 4 must be met.

Timely behavior is solved in three ways: (1) the HRT transactions commit according to a schedule and their commit times are known, (2) the commit of a finite number of SW-objects is done in finite time by reserving timeslots in the HRT-schedule and (3) the commit of SRT-objects is scheduled in the time slots left by the HRT transactions and no guarantee can be given for the timeliness requirement of the Commit. After the execution of the Commit procedure, all participants will have set the same value of $\mathcal{T}_c$ in the versions associated with the transaction.

10.3.1 HRT Commit

The example of OCC-NVTI is taken. In Fig. 22 pseudo code for the implementation of a HRT-Commit procedure is indicated. The Commit procedure for the other CCA's is similar but for the specified A-Commit parameters. At every processor one process executes the Procedure Wait-Mess. This process permanently listens for Commit messages sent by coordinators. When a Commit message is received, the Commit is executed for all data-items owned by the committing transaction

```
PROCEDURE HRT-Commit( N: set transactions, $\mathcal{T}_s$,$\mathcal{T}_e$: Timestamp, $C_f$: Counter)
BEGIN
                    $\mathcal{T}_e$ := Timestamp(time + delay)
                    Multicast( $N \oplus \mathcal{T}_e \oplus C_f$, participants)
END

PROCEDURE Wait-Mess()
VAR
        $D_p$ : data-items on processor p
BEGIN
        WHILE TRUE DO
              ON reception of " $N \oplus \mathcal{T}_e \oplus C_f$ "
                    $\forall T \in N : \forall X \in T.I \cap D_p$: A-Commit($X, T.\mathcal{T}_s, \mathcal{T}_e, C_f$)
        OD
END
```

Figure 22: HRT-Commit algorithm

It is assumed that HRT transactions are scheduled such that their interleaving always generates view-serializable schedules. When a component of a HRT system fails, often the schedule cannot be met and whole system shuts down or fails. By rendering the components very reliable the probability of such an event is made acceptably small. A hypothesis must be formulated about the consequences of a component failure.

Hypothesis 7 *When a HRT component fails, the whole system fails.*

The correctness of the protocol of Fig. 22 is rather trivially proven.

Theorem 10 *The procedures HRT-Commit and Wait-Mess together satisfy the five commit requirements and hypotheses 2 and 3.*

Proof. *Unanimity*, *Integrity*, *Validity*, *Ordering* and *Timeliness*: they are based on Multicast properties and availability of coordinator and participants.

Hypothesis 2: this is assured by the HRT schedule. The schedule should be constructed such that A-Validate always succeeds. In that case, the Validate Procedure is empty and the atomicity of A-Validate, A-Commit action pairs is assured.

Hypothesis 3: from the timeliness property it is known that the commit protocol is bounded. Equate the constant Δ_c with the time from the determination of the timestamp to the termination of the last Commit action. Take the "delay" in the setting at line 2 of $\mathcal{T}_e$ equal to Δ_c. The timestamp, $\mathcal{T}_s = C_p(t_s) \circ p$, of transaction T_{ro} is generated at local time $C_p(t_s)$ on processor p. The commit timestamp, $\mathcal{T}_e = (C_q(t_e) + \Delta_c) \circ q$ of transaction T_w is generated at processor q at local time $C_q(t_e)$. T_{ro} reads a data-item, X, located at processor d, at local time $C_d(t_r)$. Due to the communication delay $C_p(t_s) < C_d(t_r)$. A version of data-item, X, is committed by T_w at time $C_d(t_e)$. From the timeliness property of the multicast and the choice of the value of "delta" $C_q(t_e) + \Delta_c \geq C_d(t_e)$. The monotonicity of the clock function implies $t_e > t_r \Leftrightarrow C_d(t_e) > C_d(t_r)$. It follows that for a RO-transaction T_{ro} reading X at time t_r and a transaction T_w committing at time $t_e > t_r : C_q(t_e) + \Delta_c \geq C_d(t_e) > C_d(t_r) > C_p(t_s) \Rightarrow T_{ro}.\mathcal{T}_s \leq T_w.\mathcal{T}_e$, which concludes the proof. □

10.3.2 HRT/SRT Commit

When a SRT transaction reads a value produced by a HRT transaction, the TR-Read procedure is used. The commit of the HW-objects by the HRT transactions can be done with above described HRT-commit protocol (see Fig. 22). The proof of hypothesis 4 for a SRT-transaction reading a data-item from the set HW-objects is similar to the proof of hypothesis 3 in the proof of theorem 10. The commit of the HW-objects by the HRT transactions can be done with above described HRT-commit protocol.

Data-items from the set of SW-objects can be read and written by SRT transactions and read by HRT transactions. When a root SRT-transaction writes an item to be read by a HRT-transaction, another commit protocol is needed than the one used for HRT because the timeliness of the underlying SRT multicast can no longer be guaranteed. The timeliness of the commit of the SW-objects is essential. Without special measures the commit protocol can be preempted. In such a case, part of the involved objects has the state *commit* and another part is left with the state *tentative*. The verification of these unwanted consequences by the HRT transactions takes a lot of time from the HRT-schedule which can be spent differently. The timeliness of the Commit can be guaranteed by executing the Commit protocol for one or more SRT-transactions in sufficiently large and frequent gaps in the HRT schedule.

To guarantee the timeliness of the commits involving SW-objects, one process per processor executes the COMPROC Procedure in a repetitive gap of the HRT-schedule (see Figs 23 and 24). This process has access to all data-items connected to the processor. The coordinators of completed SRT-transactions fill a queue, *RQ*, at the coordinator's processor with their commit requests. A time-out is associated with each request. When the gap is met in the schedule, all processors execute the COMPROC procedure.

2PL & MVTSO The requests which exceed their deadline are removed from *RQ* (line 15). A limited amount of requests is read from the queue and stored in *nr* (line 16).

All processors multicast their requests to all correct processors, where they are received in *nr* (lines 17-19). The commits of the transactions are executed in the same order (in this case the time-out order) for the data-items associated with the executing processor (line

```
1 VAR
2
3 $RS = \{pid \mid pid$ is identifier of correct processor$\}$
4 $RQ = \{ < \mathcal{T}, t, ws, rs > \mid \mathcal{T}$ is identifier of currently active transaction with time-out value $t$ and write(read)-set $ws(rs)\}$
5 $D_p = \{ q \mid q$ is data-item resident on processor $p$ $\}$
6
7 PROCEDURE COMPROC()
8 VAR
9         nr : set of requests;
10        rqst : request;
11        cntr : Integer; $\mathcal{T}_c$ : timestamp
12
13 BEGIN
14        WHILE ingap DO
15                $\forall rq \in RQ$ DO IF $rq.t \leq endgap$ THEN $RQ := RQ \backslash rq$ FI OD;
16                $nr := \{rq \mid rq \in RQ \wedge \mid nr \mid < K \wedge rq.t > endgap\}$;
17                Multicast( nr, RS)
18                $nr := \emptyset$; $cntr := 1$;
19                $\forall p \in RS$ DO $nr := nr \cup receive(p)$ OD
20                $\mathcal{T}_c := timestamp(endgap)$;
21                WHILE $nr \neq \emptyset$ DO
22                        $rqst.t := (\mathbf{min}\ rq \in nr : rq.t)$; $nr := nr \backslash rqst$;
23                        $cntr := cntr + 1$; $\forall d \in ((rqst.ws \cup rqst.rs) \cap D_p)$ DO A-Commit(d, rqst.$\mathcal{T}$, $\mathcal{T}_c \circ cntr$) OD
24        OD; OD
25 END
```

Figure 23: SW-Commit process COMPROC for 2PL & MVTSO

22-23). All SRT-transactions verify that their requests have been executed in time. When the time-out expires, they are aborted following the SRT-commit protocol rules described in section 10.3.3.

OCC-BV The A-Commit action needs some modifications with respect to the original specification. In contrast to former requirements, a transaction that is committed in gap g can afterwards still be aborted in the same gap g. A transaction that is committed in gap g cannot be aborted in gap $g + k$ with $k > 0$.

When the gap is met in the schedule, all processors execute the COMPROC procedure (see Fig. 24). The requests which exceed their deadline are removed from RQ (line 16). A limited amount of requests is read from the queue and stored in nr (line 17).

All processors multicast their requests to all correct processors, where they are received in lr (lines 18-19). A-Validate is executed for all requests on the local data-items. When no data exist locally, A-Validate returns TRUE. When A-Validate succeeds, A-Commit is executed and the requests is added to nr (line 25-26). The set of committed requests is broadcast to all participants. All participants receive the same request sets from all participants (line 28-29). For requests that are removed at at-least one processor, a local A-Abort is executed (line 30).

The later Abort removes versions that are not yet read by any other transactions. Transactions read versions that are installed in one of the earlier terminated gaps.

Theorem 11 *COMPROC satisfies the five commit requirements and hypotheses 2, 3 and 4.*

Proof. *Unanimity:* the Unanimity requirement is met by the unanimity property of the supporting multicast. All requests are sent to all participants. When a participant receives a

```
1 VAR
2
3 $RS = \{pid \mid pid$ is identifier of correct processor$\}$
4 $RQ = \{ < \mathcal{T}_i, t, ws, rs, \mathcal{T}_e > \mid \mathcal{T}_i$ is identifier of currently active transaction with time-out value $t$ and write(read)-set $ws(rs)\}$
5 $D_p = \{ q \mid q$ is data-item resident on processor $p$ $\}$
6
7 PROCEDURE COMPROC()
8 VAR
9           tr, nr, lr : set of requests;
10          rq : request;
11          ws1 : set of data-items;
12          b: boolean, cntr: integer;
13          $\mathcal{T}_c$ : Timestamp;
14 BEGIN
15          WHILE ingap DO
16                  $\forall rq \in RQ$ DO IF $rq.t \leq endgap$ THEN $RQ := RQ \backslash rq$ FI OD;
17                  $nr := \{rq \mid rq \in RQ \wedge \mid nr \mid < K \wedge rq.t > endgap\}$;
18                  Multicast( nr, RS)
19                  $lr := \emptyset; \forall p \in RS$ DO $lr := lr \cup receive(p)$ OD
20                  $tr := lr$; $nr := \emptyset$; $cntr := 1$
21                  $\mathcal{T}_c := timestamp(endgap)$
22                  WHILE $lr \neq \emptyset$ DO
23                          $rq.t := (\mathbf{min}\ q \in lr : q.t)$; $lr := lr \backslash rq$; $b :=$ TRUE
24                          $\forall d \in ((rq.ws \cup rq.rs) \cap D_p)$ DO $b := b \wedge$ A-Validate($d$, $rq.\mathcal{T}_i, rq.\mathcal{T}_e$) OD
25                          IF b THEN $cntr := cntr+1$; $\forall d \in ((rq.ws \cup rq.rs) \cap D_p)$ DO A-Commit($d, rq.\mathcal{T}_i, rq.\mathcal{T}_c \circ cntr$)
26                                  $nr := nr \cup \{rq\}$; FI
27                  OD; OD
28                  Multicast( nr, RS)
29                  $lr := \emptyset; \forall p \in RS$ DO $lr := lr \cup receive(p)$ OD
30                  $\forall rq \in tr - lr$, $\forall d \in (rq.ws \cap D_p)$ DO A-Abort($d$, $rq.\mathcal{T}_i$) OD
31          OD
32 END
```

Figure 24: SW-Commit process COMPROC for OCC-BV

multicast message, all participants receive it. At line 19 of both COMPROC procedures, all processors have the same commit requests in the variable lr. In case of OCC-BV, Validate requests are executed in the order determined by the time-out values at line 24 and all processors execute the Validate requests in the same order. When a transaction cannot be validated, the request is not added to nr. All processors send the contents of nr to all processors. From the unanimity of the multicast it follows that all processors receive the same values and end up with the same requests in lr at line 29. At all processors the same transactions are aborted at line 30. At line 25 all remaining data-items of a given transaction are committed with the same $\mathcal{T}_e$ value. This concludes the unanimity part.

Integrity: in case of 2PL & MVTSO, all participants agree to commit all received requests due to hypothesis 7 and the requirement is met for all requests. In case of OCC-BV, participants only agree when A-Validate returns TRUE at line 22. Refused requests are not inserted into nr. All participants send each other their nr lists and abort all transactions that are refused by any processor. Agreement depends on the former commits. The removal of a committed version in a given gap g does not change the agreement of later transactions committed in the same gap g.

Validity: in case of 2PL & MVTSO all participants agree to participate in the Commit due to hypothesis 7 and their requests to commit are stored in nr. Once stored in nr, the Commit acctions are executed at line 23. In case of OCC-BV, the requests validated by all participants are maintained in *nr*. All participants receive the same lists of validated requests. When all participants agree, their requests are maintained in *lr* at line 28. For those requests no Aborts are executed.

Ordering: the Ordering is trivially met by choosing the commit time at the end of the gap and postfixing the value of counter *cntr*.

Timeliness: the Timeliness property of the underlying multicast assures that messages are sent within bounded time. From the moment that the request is transmitted, all participants will receive this request in bounded time. They only receive a bounded number of requests and will send a bounded number of commit requests to the participants. Within bounded time the participants receive the message and will execute the commit. Consequently, the timeliness requirement is met.

Hypothesis 2: the proof of hypothesis 2 is separated in two parts.

Exception atomicity: for each transaction either an Abort or a Commit action is executed. After the execution of COMPROC all versions belonging to a given transactions have the state *commit* or have been removed.

Concurrency atomicity: an order is established by the time-out values. . The A-Validate and A-Commit actions are sequentially executed in this same order on all processors.

Hypotheses 4 and 3: new HRT or SRT transactions will start after the gap and will find the appropriate versions ready and committed. Tentative versions will be committed in one of the following gaps with a timestamp larger than the starting times of the active transactions.
□

The COMPROC procedure for OCC-BV leads to unwanted Aborts. A transaction T_1 that aborts on site p and but writes an item on site q may lead to the abort of a transaction T_2 that read q. The later global abort of T_1 permits the Commit of T_2 that is not performed in COMPROC. Remedies to this situation can be envisaged by sending information about all accessed data by all committable transactions to all sites. This additional information can be used to construct a commit order that allows the minimum number of aborts. However, such a strategy leads to many messages.

In case of OCC-NVTI, the determination of a globally correct C_f value leads to the exchange of even more information. In this case it is probably wiser to execute the Validate and Commit pairs one after the other and not group them as is the case in COMPROC.

10.3.3 SRT Commit

The example of OCC-NVTI is taken. Other CCA's can be treated similarly. The SRT-Commit (shown in figs 26 and 25) depends on the clock-synchronization and the Multicast facilities. The major difference with the former two protocols is the invalidity of the timeliness property of the underlying multicast protocol. Therefore, extra measures must be taken to validate hypothesis 3. A "validate" message is sent to all participants. Each participant calculates an appropriate commit timestamp based on its local clock. The transaction identifier with the calculated timestamp is stored in the history $H : identifier \rightarrow Timestamp$. Each participant initiates a time-out $> Time + delay$ and returns the calculated Commit timestamp. When all results are returned and validated, a "commit" together with the maximum of the received timestamps is sent by the coordinator. Commits that take too long are aborted. The timelines of the "Commit" is verified by each participant. When a participant sees that the time-out has passed, it sends an "abort" message to all participants including itself. Because all multicast messages are ordered, all participants will receive the same messages in the same order and will execute either A-Abort or A-Commit. After arrival of the "commit" message, the entry in H is removed and the commit is executed on all data-items belonging to T. On arrival of an "abort" message the corresponding item is removed from H and the abort is executed on the with T associated data-items. When a sequence of messages arrives for one transaction, the first request specified in the message is executed and the transaction identifier is removed from H, after which all messages for this transaction are ignored.

```
VAR
        $H$ : identifier $\rightarrow$ Timestamp
        $D_p = \{ q \mid q$ is data-item resident on processor $p \}$

PROCEDURE Wait-Mess()
BEGIN
        $H := \emptyset$
        WHILE TRUE DO
                ON reception of $I \oplus \mathcal{T}_i \oplus C_f \oplus C_l \oplus$ "validate"
                        set Time-Out;
                        $\forall v \in I \cup D_p$ DO A-Validate($v$, $\mathcal{T}_i$, $C_f$, $C_l$) OD
                        $H := H \cup \{\mathcal{T}_i \mapsto TS(Time + delay)\}$
                        send( $\mathcal{T}_i \oplus TS(Time + delay)$)
                ON reception of $I \oplus \mathcal{T}_i \oplus \mathcal{T}_e \oplus C_f \oplus$ "commit"
                        IF $\mathcal{T}_i \in dom(H)$ THEN
                                $dom(H) := dom(H) - \{\mathcal{T}_i\}$
                                $\forall v \in I \cap D_p$ DO A-Commit($v$, $\mathcal{T}_i$, $\mathcal{T}_e$, $C_f$) OD
                        FI
                ON reception of $I \oplus \mathcal{T}_i \oplus$ "abort"
                        IF $\mathcal{T}_i \in dom(H)$ THEN
                                $\forall v \in I \cap D_p$ DO A-Abort( $v$, $\mathcal{T}_i$) OD
                                $dom(H) := dom(H) - \{\mathcal{T}_i\}$
                        FI
                ON occurrence of "time-out"
                        Multicast($I \oplus \mathcal{T}_i \oplus$ "abort", participants)
        OD
END
```

Figure 25: Wait-Mess for SRT-Commit

The precondition of the RO-Read action must be extended with:

$$\mathcal{T} < (\mathbf{min}\; s \in dom(H) : H(s)) \tag{77}$$

to assure that hypothesis 3 is met. The RO-Read action may only be started when no transaction waits to be committed with a $\mathcal{T}_e$ smaller than the $\mathcal{T}_s$ of the RO-Read invoking RO-transactions.

Theorem 12 *The procedures SRT-Commit, SRT-Validate and Wait-mess satisfy the first four commit requirements and hypotheses 2 and 3.*

Proof. *Unanimity:* the Unanimity requirement is met by the unanimity and ordering properties of the underlying multicast. All correct participants will receive the same multicast messages in the same order. The first decision on a given transaction is received by all participants and consecutively executed. All participants will take the same decision for any

```
PROCEDURE SRT-Commit( I: data-items, $\mathcal{T}_s, \mathcal{T}_e$: Timestamp, $C_f$: Counter)
BEGIN
        Multicast( $I \oplus \mathcal{T}_s \oplus \mathcal{T}_e \oplus C_f \oplus$ "commit", participants)
END

PROCEDURE SRT-Validate(I: data-items, $\mathcal{T}_s, \mathcal{T}_e$ : Timestamp, $C_f, C_l$ : Counter)
VAR
        $C_b, C_e$ : Counter
        $\mathcal{T}$: Timestamp
BEGIN
        Multicast( $I \oplus \mathcal{T}_s \oplus C_f \oplus C_l \oplus$ "validate", participants)
        $\mathcal{T}_e := 0$
        $\forall$ participants DO receive($\mathcal{T}$); $\mathcal{T}_e := \max(\mathcal{T}, \mathcal{T}_e)$ OD
END
```

Figure 26: SRT-Commit algorithm

given transaction. Once a A-Commit or A-Abort has been executed for a transaction T, this transaction is removed from H. Later messages in connection with transaction T are ignored.

Integrity: participants only receive "commit" requests from the coordinator when all participants have returned an answer, implying their agreement.

Validity: when no failures or time-outs occur, all participants will send their answers to the coordinator. The "commit"message of the coordinator arrives at all destinations after which A-Commit is executed.

Ordering: See proof of lemma 17.

Hypothesis 2: follows directly from proof of lemma 17.

Hypothesis 3: the RO-Read precondition assures that when RO-Read is executed at local time t with a given $\mathcal{T}_s$, all committing transactions will have a $\mathcal{T}_e < \mathcal{T}_s$. After t, commit requests are added to H with a $\mathcal{T}_e$ that is larger than $\mathcal{T}_s$. Consequently, all transactions committing at local time $t' > t$ have a $\mathcal{T}_e > \mathcal{T}_s$. □

Transactions can be aborted when the precondition of A-Validate is not met because another transaction has put the state of the specified item X equal to *validating*, or the "validate" request can be put in a wait-queue till the state of the data-item has changed. In the latter case, the total ordering of all multicast messages prevents deadlocks. When a transaction T_1 waits for the commit of another transaction T_2, The validate request of T_1 was sent before the request of T_2. The total ordering of the "validate" requests prevents cycles in the wait-for graph.

11 Implementation restrictions

11.1 finite number of versions

In the above it has been assumed that once created versions remain permanently available. This is not realistic and a fixed number of versions per object is assumed.

The conditions under which a RO-transaction based on A-Read outperforms the RO-transaction based on the RO-Read action need to be investigated.

RO-transactions can be aborted for two reasons: (1) missing of the deadline and (2) un-

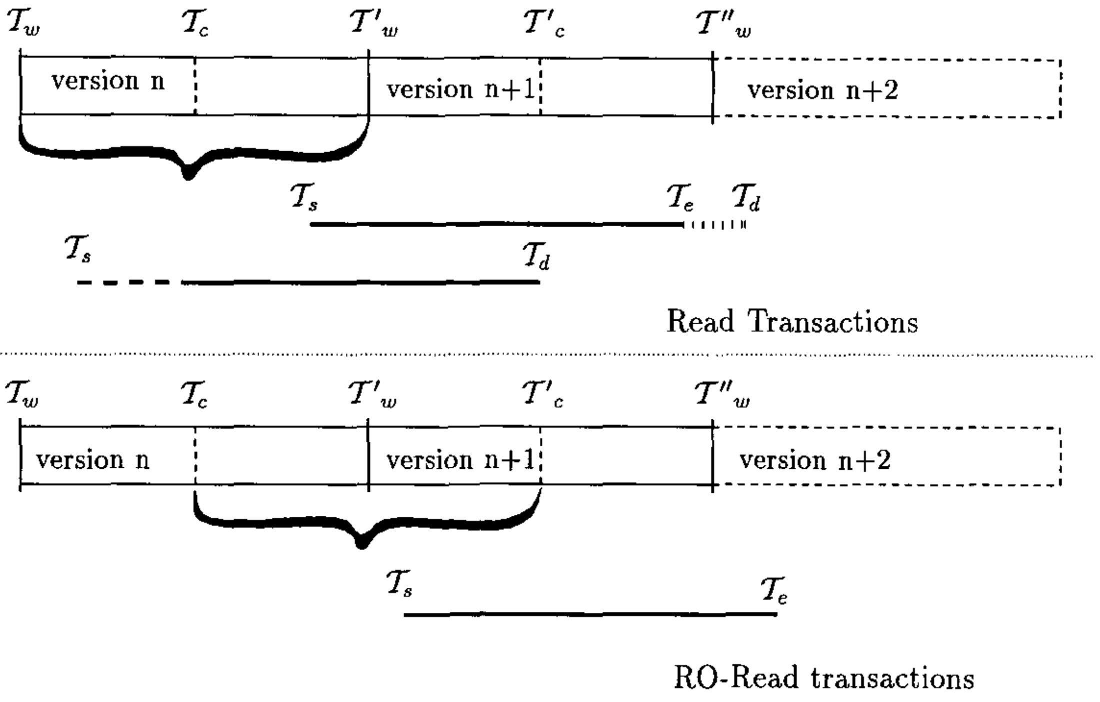

Figure 27: RO-Read versus Read with two versions

availability of the required version. In Fig. 27, the different behaviours are shown schematically. Time progresses from left to right. In the upper part the A-Read action behaviour and in the lower part the RO-Read action behaviour is shown. The evolution of the versions are shown by the sequence of rectangles and the evolution of the transactions by the full and dotted lines. At time T_w version n of a given item is created. At time T_c the version is committed and from the following time T'_w the versions $n+1$ and $n+2$ are created. For this example it is assumed that only two versions are kept per item. Therefore, version $n+2$ is drawn with a dotted line because at its creation time version n is removed. When A-Read is used (upper part of figure), transactions which are started between the creation times, $T_w < T_s < T'_w$, of version n and $n+1$ will select version n. When version n is selected at the beginning of the transaction, the version can only be accessed after time T_c. In the Figure this is shown with the dotted line. The transaction can be so delayed that it meets its deadline before it terminates and is aborted. The transaction started with $T_c < T_s < T'_w$, will not wait and terminate before its deadline (shown by the dotted line between T_e and T_d).

In the lower part of the figure the behaviour of the RO-Read actions is visualized. A transaction starting between the commit-times, $T_c < T_s < T'_c$, of version n and version $n+1$ selects version n. Transactions will not wait and terminate correctly assuming that they are not preempted. This is shown by the uninterrupted line at the bottom of the figure. When this transaction accesses version n at the end of the transaction, the version has been replaced by version $n+2$ and the transaction needs to be aborted.

Performance in the case of the A-Read action is limited by the longer duration of the transaction and in the case of the RO-Read actions is limited by the shorter availability of a given version for the transactions. A criterion can be established when one method will outperform the other.

Assume that k versions of all items are regularly updated with a fixed update interval θ:

$$\theta = T'_w - T_w = T'_c - T_c \tag{78}$$

It is assumed that the distribution, $P(T_s)$, of start times of transactions which need version n is flat.

$$\begin{array}{llll} & & \text{A-Read} & \text{RO-Read} \\ P(T_s) = 1/\theta & \text{if} & T_w < T_s < T'_w & T_c < T_s < T'_c \\ P(T_s) = 0 & \text{otherwise} & & \end{array} \tag{79}$$

The probability, $P(T_s < X)$, that T_s has a value lower than a given value, X, is given by:

$$P(T_s < X) = \int_{-\infty}^{X} P(T_s)\, dT_s \quad \begin{array}{llll} \text{A-Read} & & \text{RO-Read} & \\ = 0 & X < T_w & = 0 & X < T_c \\ = \frac{X-T_w}{\theta} & T_w \leq X < T'_w & = \frac{X-T_c}{\theta} & T_c \leq X < T'_c \\ = 1 & X \geq T'_w & = 1 & X \geq T'_c \end{array} \tag{80}$$

When an item is read with A-Read, the selected version may be committed or still tentative. The probability, $P(w_1 > 0)$, that the transaction needs to wait for time, w_1, on the first item is given by the probability that $T_s < T_c$:

$$P(w_1 > 0) = P(T_s < T_c) = \frac{T_c - T_w}{\theta} \quad \text{if } T_w \leq T_c < T'_w \tag{81}$$

Assume that the reading of a data-item takes a fixed time, δ (including eventual preemption times), and that each transaction reads a fixed number, n, of data-items. The transaction execution time, ΔT, is then given by:

$$\Delta T = n.\delta \tag{82}$$

It is possible that the reading of a data-item is shorter than the time between the start and commit of a transaction;

$$\delta < T_c - T_w \tag{83}$$

Assume that the creation times of versions of different data-items are not related. In that case after the access of the first data-item the transaction may wait for the commit of the second data-item and after the access of the n^{th} for the commit of the $(n+1)^{th}$. $P_1(w)$ is the probability that a transaction waits an interval of length w_1 for the commit of item 1. $P_1(w)$ is equal to the probability $P(T_c - T_s = w_1)$ if $T_s < T_c$ and is equal to the probability $P(w_1 = 0) = P(T_s \geq T_c)$:

$$P_1(w) = P(w_1 = T_c - T_s \mid T_s < T_c) + P_1(w_1 = 0 \mid T_s \geq T_c) \tag{84}$$

This can be rewritten as:

$$P_1(w) = P(w_1) + P_1(0) \tag{85}$$

where each term is given by:

$$\begin{aligned} P_1(0) = P(T_s > T_c) &= \int_{T_c}^{T'_w} 1/\theta \, dT_s = \tfrac{T'_w - T_c}{\theta} && \text{if } w_1 = 0 && (86)\\ P(w_1) &= 1/\theta && \text{if } 0 < w_1 \leq T_c - T_w && (87)\end{aligned}$$

$P_2(w)$ is the probability that a transaction waits a time interval w_2 for the commit of item 2. $P_2(w)$ can be calculated with the aid of $P_1(w)$. The probability, $P_2(w)$, must be separated in two parts: (1) when $T_s+\delta+w_1 > T_c$, then $P_2(w)$ is determined by the probability distribution $P_1(w)$ and (2) when $T_s+\delta+w_1 \leq T_c$, then $P_2(w)$ is determined by the probability that $T_c - T_s - \delta = w_2$ for item 2:

$$P_2(w) = P_2(w \mid T_s + w_1 + \delta > T_c) + P_2(w \mid T_s + w_1 + \delta \leq T_c) \tag{88}$$

Each of these terms must be considered for the case that $w_1 = 0$ and the case $w_1 > 0$.

$$\begin{aligned} &P_2(w \mid T_s + w_1 + \delta > T_c) = \\ &\qquad P_2(w \mid T_s + \delta > T_c \wedge w_1 = 0) + P_2(w \mid T_s + w_1 + \delta > T_c \wedge w_1 > 0) && (89)\\ &P_2(w \mid T_s + w_1 + \delta \leq T_c) = \\ &\qquad P_2(w \mid T_s + \delta \leq T_c \wedge w_1 = 0) + P_2(w \mid T_s + w_1 + \delta \leq T_c \wedge w_1 > 0) && (90)\end{aligned}$$

Each of these terms can be calculated:

$$\begin{aligned} P_2(w \mid T_s + \delta > T_c \wedge w_1 = 0) &= P_1(0).P(T_s + \delta > T_c) = \\ P_1(0). \int_{T_c-\delta}^{T'_w} 1/\theta \, dT_s &= \tfrac{(T'_w - T_c)(T'_w - T_c + \delta)}{\theta^2} \qquad \text{if } w = 0 \end{aligned} \tag{91}$$

$$\begin{aligned} P_2(w \mid T_s + w_1 + \delta > T_c \wedge w_1 > 0) &= P(w_1).P(T_s + w_1 + \delta > T_c) = \\ P(w_1). \int_{T_c-\delta-w_1}^{T'_w} 1/\theta \, dT_s &= \tfrac{T'_w - T_c + \delta + w}{\theta^2} \qquad \text{if } 0 < w < T_c - T_w \end{aligned} \tag{92}$$

$$\begin{aligned} P_2(w \mid T_s + \delta \leq T_c \wedge w_1 = 0) &= \\ P_1(0).P(w_2 = T_c - T_s - \delta) &= (T'_w - T_c)/\theta^2 \quad \text{if } 0 < w < T_c - T_w - \delta \end{aligned} \tag{93}$$

$$\begin{aligned} P_2(w \mid T_s + w_1 + \delta \leq T_c \wedge w_1 > 0) &= \\ \int_0^w P(w_1) \, dw_1 P(w_2 = T_c - \delta - T_s) &= w/\theta^2 \quad \text{if } 0 < w < T_c - T_w - \delta \end{aligned} \tag{94}$$

The wait distribution after two items, under the condition: $(T_w + \delta < T_c)$, is given by:

$$P_2(w) \begin{aligned} &= (T'_w - T_c)(T'_w - T_c + \delta)/\theta^2 && \text{if } w = 0 \\ &= (2(w + T'_w - T_c) + \delta)/\theta^2 && \text{if } 0 < w < T_c - T_w - \delta \\ &= (T'_w + \delta + w - T_c)/\theta^2 && \text{if } T_c - T_w - \delta \leq w < T_c - T_w \end{aligned} \tag{95}$$

The wait distribution after i items, under the condition: $(T_w + (i-1).\delta < T_c)$, is given by:

$$P_i(w) = \begin{cases} \prod_{m=0}^{i-1}(T'_w - T_c + m.\delta)/\theta & \text{if } w = 0 \\ \sum_{k=0}^{i-1} A_{i,k,i} w^k & \text{if } 0 < w < T_c - T_w - (i-1).\delta \\ \sum_{k=0}^{i-1} A_{i,k,i-1} w^k & \text{if } T_c - T_w - (i-1).\delta \le w < T_c - T_w - (i-2).\delta \\ . & \\ . & \\ \sum_{k=0}^{i-1} A_{i,k,1} w^k & \text{if } T_c - T_w - \delta \le w < T_c - T_w \end{cases} \tag{96}$$

The coefficients $A_{i,k,j}$ will be recursively defined. Again separate the probability of waiting on the i^{th} item in two parts:

$$P_i(w) = P(w \mid T_s + w_{i-1} + (i-1).\delta > T_c) + P_i(w \mid T_s + w_{i-1} + (i-1).\delta \le T_c) \tag{97}$$

Each of these terms must be considered for the case that $w_{i-1} = 0$ and the case $w_{i-1} > 0$.

$$P_i(w \mid T_s + w_{i-1} + (i-1).\delta > T_c) = \tag{98}$$
$$P_i(w \mid T_s + (i-1).\delta > T_c \wedge w_{i-1} = 0) + P_i(w \mid T_s + w_{i-1} + (i-1).\delta > T_c \wedge w_{i-1} > 0)$$
$$P_i(w \mid T_s + w_{i-1} + (i-1).\delta \le T_c) = \tag{99}$$
$$P_i(w \mid T_s + (i-1).\delta \le T_c \wedge w_{i-1} = 0) + P_i(w \mid T_s + w_{i-1} + (i-1).\delta \le T_c \wedge w_{i-1} > 0)$$

Each of these terms can be calculated:

$$\begin{array}{ll} P_i(w \mid T_s + (i-1).\delta > T_c \wedge w_{i-1} = 0) & = P_{i-1}(0).P(T_s + (i-1).\delta > T_c) = \\ P_{i-1}(0).\int_{T_c-(i-1).\delta}^{T'_w} 1/\theta \, dT_s & = P_{i-1}(0).(T'_w - T_c + (i-1).\delta)/\theta = \\ 1/\theta^i.\prod_{m=1}^{i}(T'_w - T_c + (m-1).\delta) & \text{if } w = 0 \end{array} \tag{100}$$

$$\begin{array}{ll} P_i(w \mid T_s + w_1 + (i-1).\delta > T_c \wedge w_{i-1} > 0) & = P(w_{i-1}).P(T_s + w_{i-1} + (i-1).\delta > T_c) = \\ P(w_{i-1}).\int_{T_c-(i-1).\delta-w_{i-1}}^{T'_w} 1/\theta \, dT_s & = \\ P_{i-1}(w).(T'_w - T_c + (i-1).\delta + w)/\theta & \text{if } 0 < w < T_c - T_w \end{array} \tag{101}$$

$$\begin{array}{ll} P_i(w \mid T_s + (i-1).\delta \le T_c \wedge w_{i-1} = 0) & = \\ P_{i-1}(0).P(w_i = T_c - T_s - (i-1).\delta) & = P_{i-1}(0)/\theta = \\ (1/\theta^i).\prod_{m=1}^{i-1}(T'_w - T_c + (m-1).\delta) & \text{if } 0 < w < T_c - T_w - (i-1).\delta \end{array} \tag{102}$$

$$\begin{array}{ll} P_i(w \mid T_s + w_{i-1} + (i-1).\delta \le T_c \wedge w_{i-1} > 0) & = \\ \int_0^w P(w_{i-1}) \, dw_{i-1} P(w_i = T_c - (i-1).\delta - T_s) & = \\ 1/\theta.\int_0^w P(w_{i-1}) \, dw_{i-1} & \text{if } 0 < w < T_c - T_w - (i-1).\delta \end{array} \tag{103}$$

From the above equations the values of $A_{i,k,j}$ can be determined:

if $1 \leq j < i \wedge i < k < 0$

$$A_{i,k,j} = A_{i-1,k,j}.(T'_w - T_c + (i-1).\delta)/\theta + A_{i-1,k-1,j}/\theta \tag{104}$$

if $1 \leq j < i \wedge k = 0$

$$A_{i,0,j} = A_{i-1,0,j}.(T'_w - T_c + (i-1).\delta)/\theta \tag{105}$$

if $j = i \wedge i < k < 0$

$$A_{i,k,i} = A_{i-1,k,i}.(T'_w - T_c + (i-1).\delta)/\theta + A_{i-1,k-1,i}/\theta + A_{i-1,k-1,i}/(k.\theta) \tag{106}$$

if $j = i \wedge k = 0$

$$A_{i,0,i} = A_{i-1,0,i}.(T'_w - T_c + (i-1).\delta)/\theta + (1/\theta^i).\prod_{m=1}^{i-1}(T'_w - T_c + (m-1).\delta) \tag{107}$$

The mean duration, $\Delta T'$, of a transaction which uses A-Read and waits for the versions to commit is determined by:

$$\Delta T' = \Delta T + \int_0^{T_c - T_w} w\, P_n(w)\, dw \tag{108}$$

Next the probability must be calculated that a transaction, using A-Read, is aborted because its waiting time is unacceptably long. Transactions can only be aborted when the worst case wait-time, $T_c - T_w$, plus the total execution time , ΔT, is larger than the for execution allocated time period $T_d - T_s$:

$$T_d - T_s < \Delta T + T_c - T_w \tag{109}$$

The maximum time, W, that a transaction, using A-Read, may wait is defined by:

$$W = T_d - T_s - \Delta T > 0 \tag{110}$$

Negative values of W imply that all transactions will be aborted. The probability that a transaction which uses A-Read, executes beyond its deadline is given by: $P_n(w > W)$:

$$\begin{array}{ll} P_n(w > W) & = \int_W^\infty P_n(w)dw \quad \text{if } 0 \leq W < T_c - T_w \\ & = 0 \qquad\qquad\qquad\;\; \text{otherwise} \end{array} \tag{111}$$

The chances $Pf(Pf')$ that a transaction using A-Read(RO-Read) cannot find the appropriate version of the last, k^th, item is calculated now. The probability, Pf', of abort of a transaction using Ro-Read is equal to the probability $P(T_s + \Delta T > T_w + k.\theta)$:

$$Pf' = \int_{T_w + k.\theta - n.\delta}^{\infty} P(T_s)\, dT_s = \begin{array}{ll} 0 & \text{if } T_w + k.\theta - n.\delta > T'_c \\ \frac{T'_c - T_w - k.\theta + n.\delta}{\theta} & \text{if } T_c < T_w + k.\theta - n.\delta \leq T'_c \\ 1 & \text{if } T_c \geq T_w + k.\theta - n.\delta \end{array} \tag{112}$$

The probability, Pf, that under the same circumstances a transaction does not find the appropriate version using A-Read is equal to the probability $P(T_s + w + \Delta T > T_w + k.\theta)$. This is equivalent to the probability $P_n(w > T_w + k.\theta - T_s - n.\delta)$ for all possible values of T_w.

$$Pf = \int_{T_w}^{T'_w} P(T_s)\, dT_s \int_{T_w+k.\theta-T_s-n.\delta}^{\infty} P_n(w)\, dw \tag{113}$$

Assume that the aborts due to the unavailability of versions is independent of the aborts due to missing of deadlines. The RO-Read actions will result in less aborts than the Read actions when

$$(1 - P_n(w > W))(1 - Pf) < 1 - Pf' \tag{114}$$

11.2 Implementation of RO-Read Action

The RO-Read implementation for SRT transactions is straightforward in the case of OCC-BV and 2PL. Care must be taken in the case of MVTSO and OCC-NVTI, as the value of $\mathcal{M}_{ro}$ must be determined in an efficient way. Three possibilities exist: (1) requesting the values from the provider when needed in two phases, (2) requesting the values from the provider in one phase and (3) regularly providing the values to requester at the moment they are available. The third option is a valid one for Real-Time systems which have a periodic behaviour and often transactions require values on a repetitive basis.

11.2.1 2 phase communication

The RO-transaction as shown in figures 3, 12 and 13 need to be refined to calculate the $\mathcal{M}_{ro}$ value. The algorithm for requesting a value in two phases is shown in fig. 28. Two functions *loc* and *ident* are introduced. The function *loc* returns the identifier of the processor where data-item X is located, and *ident* returns the identifier of data-item X.

MVTSO The function Close, with version parameter v, assures that for all versions, x, with $x.\mathcal{T}_w < v.\mathcal{T}_w$: the read timestamp, $\mathcal{T}_r$ is equal to the write Timestamp $\mathcal{T}_w$ of its immediate successor. Consequently, no more versions can be added between the versions created before version v.

In the RO-Transaction, for every required data-item, X, the minimum, $\mathcal{T}_m$, is determined of the not yet committed versions and the maximum of the $\mathcal{T}_w$ values of the committed versions, x, with $x.\mathcal{T}_w < \mathcal{T}_m$ is returned. At line 36 these $\mathcal{T}_m$ values are stored in the set ST. For all returned data-item, X, no new version, x, can be created with a $x.\mathcal{T}_w < \mathcal{T}_m$ because all $\mathcal{T}_r$ values are made equal to all the $\mathcal{T}_w$ values of their immediate successors.

At line 38, for all data-items, X, RO-Read is invoked with a parameter, $\mathcal{M}_{ro}$, which is equal to the minimum of the set ST. The returned values are stored in the local variables $A[X]$ at line 39. At every processor the Procedure Wait-Mess is permanently waiting to receive the messages of the RO-transactions.

OCC-NVTI In the RO-Transaction, for every required data-item, X, the minimum, $\mathcal{C}_m$, is determined of the not yet committed versions and the versions with a $\mathcal{T}_e$ value larger than $\mathcal{T}_s$. The minimum of $\mathcal{C}_m$ and the maximum of the $\mathcal{C}_b$ values of the committed versions, x, with $x.\mathcal{C}_b < \mathcal{C}_m$ is returned. At line 30 of Fig. 29 these $\mathcal{C}_m$ values are stored in the set SC. The calculated $\mathcal{C}_m$ value stays correct because new versions are created by A-Write with $\mathcal{C}_b$ values larger than all existing ones.

```
FUNCTION Close( v : version)
{PRE:   $(\exists X \in D : v \in X.V) \wedge x_m.\mathcal{T}_w = (\mathbf{max}\ x \in X.V, x.\mathcal{T}_w < v.\mathcal{T}_w : x.\mathcal{T}_w) \wedge v.\mathcal{T}_w = T$
POST:   $x_m.\mathcal{T}_r = T \wedge Close(x_m)$
}

PROCEDURE ROtime( X: data-item): Timestamp
{PRE:   $\mathcal{T}_m = (\mathbf{min}\ x \in X.V, x.S \neq commit : x.\mathcal{T}_w)$
        $\wedge v \in X.V \wedge v.\mathcal{T}_w = (\mathbf{max}\ x \in X.V, x.\mathcal{T}_w < \mathcal{T}_m, x.S = Commit : x.\mathcal{T}_w)$
POST:   $Close(v) \wedge RET : \mathcal{T}_m$
}

PROCEDURE RO-Read( X: data-item, $\mathcal{T}$: Timestamp, var a: value)
{PRE: $v \in X.V \wedge v.\mathcal{T}_w = (\mathbf{max}\ x \in X.V, x.S = commit, x.\mathcal{T}_w \leq \mathcal{T} : x.\mathcal{T}_w) \wedge v.value = A$
POST: $a = A$
}

PROCEDURE Wait-Mess()
BEGIN
        WHILE TRUE DO
                ON reception of ident(X)
                        T := ROtime(X)
                        send( T)
                ON reception of ident(X) $\oplus$ T
                        RO-Read(X, $\mathcal{T}$, v)
                        send( v)
        OD
END

PROCEDURE RO-Transaction(RS: read actions, I: data-items, t: timestamp)
VAR
        ST : set of Timestamps, $\mathcal{T}_s, \mathcal{M}_{ro}$ : Timestamp
        A[data-items]: values
BEGIN
        $\mathcal{T}_s$ = f( t, time); ST := $\emptyset$
        $\forall act[X] \in RS$: send( loc(X), ident(X))
        $\forall act[X] \in RS$: receive( $\mathcal{T}_m$); $ST := ST \cup \mathcal{T}_m$
        $\mathcal{M}_{ro} := \min(ST)$
        $\forall act[X] \in RS$: send( loc(X), ident(X) $\oplus \mathcal{M}_{ro}$)
        $\forall act[X] \in RS$: receive( A); $A[X] := A$
        execute calculations
END
```

Figure 28: RO-Read requests for MVTSO, two communication phases

At line 32 of Fig. 29, RO-Read is invoked with a parameter, $\mathcal{M}_{ro}$ which is equal to the minimum of the set SC. The returned values are stored in the local variables $A[X]$.

In these algorithms the communication is done in two phases. In the first phase, requests for the minima are sent to all participating processors. Based on the returned values, $\mathcal{M}_{ro}$ is calculated. In the second phase the RO-Read Procedure is invoked at all participants with the communicated $\mathcal{M}_{ro}$ value. The returned values are used. This method is quite expensive in messages.

11.2.2 1 phase communication

MVTSO In Fig. 30, a one phase algorithm is shown for MVTSO. It is assumed that a lower limit on the age of the acquired values can be specified. Older values are assumed to be unusable for the application. This age parameter is sent over to the participants which

```
PROCEDURE ROtime( X: data-item, $\mathcal{T}_s$: Timestamp): Counter
{PRE:   $\mathcal{C}_m = ( min\ x \in X.V, (x.S \neq commit \vee x.\mathcal{T}_s > \mathcal{T}_e : x.C_b))$
POST:   RET: $\min(\mathcal{C}_m, (\mathbf{max}\ x \in X.V, x.S = Commit : x.C_b))$
}

PROCEDURE RO-Read( X: data-item, C: Counter, var a: value)
{PRE:   $v \in X.V \wedge (\forall x \in X.V, x.S = commit, x.C_b \leq \mathcal{C} : x.C_b \leq v.C_b) \wedge v.value = A \wedge v.C_b \leq \mathcal{C} \wedge v.S = commit$
POST:   $a = A$
}

PROCEDURE Wait-Mess()
BEGIN
        WHILE TRUE DO
                ON reception of ident(X)
                        $\mathcal{C}_m$ := ROtime(X)
                        send( $\mathcal{C}_m$)
                ON reception of ident(X) $\oplus$ C
                        RO-Read(X, C, v)
                        send( v)
        OD
END

PROCEDURE RO-Transaction(RS: read actions, I: data-items, t: timestamp)
VAR
        SC : set of Timestamps, $\mathcal{T}_s, \mathcal{M}_{ro}$ : Timestamp
        A[data-items]: values
BEGIN
        $\mathcal{T}_s$ = f( t, time); SC := $\emptyset$
        $\forall act[X] \in RS$: send( loc(X), ident(X))
        $\forall act[X] \in RS$: receive( $\mathcal{C}_m$); SC := SC $\cup\ \mathcal{C}_m$
        $\mathcal{M}_{ro} := \min(SC)$
        $\forall act[X] \in RS$: send( loc(X), ident(X) $\oplus\ \mathcal{M}_{ro}$)
        $\forall act[X] \in RS$: receive( A); $A[X] := A$
        execute calculations
END
```

Figure 29: RO-Read requests for OCC-NVTI, two communication phases

return committed versions which are younger than the specified age-limit and which are older than the oldest not yet commited version. On the base of the returned sets of versions the $\mathcal{M}_{ro}$ value can be calculated and the required values can be selected locally because they are already available at the requesting processor.

In the Procedure VClose, the minimum, $\mathcal{T}_m$, of all $\mathcal{T}_w$ values of the uncommitted versions is determined. The set of versions is determined which have a $\mathcal{T}_w$ values which lies between this minimum and the specified age-limit $\mathcal{T}$. This set is returned and the $\mathcal{T}_r$ values of the committed versions are set equal to the $\mathcal{T}_w$ values of their immediate committed successor versions. The *Close* function in the POST condition of VClose is the same as specified in Fig. 28. The Procedure RO-Read has the same specification as the RO-Read of the former algorithm with the exception that the data-item parameter is replaced by a set of versions.

Again at every processor the Wait-Mess procedure is started which waits for messages. On reception of a message, the Function VClose is invoked and the result set is sent back to the requester.

The RO-Transaction sends a request for the set of versions to all participants and waits for the results which consecutively are stored in the set XS. For every data-item the maximum of all $\mathcal{T}_w$ values of the versions is calculated. The minimum of all maxima is stored in $\mathcal{T}_m$ and

```
PROCEDURE VClose( X: data-item, $\mathcal{T}$: Timestamp) : set of versions
{PRE:   $\mathcal{T}_m = (\mathbf{min}\ x \in X.V, x.S \neq commit : x.\mathcal{T}_w) \wedge A \subseteq X.V \wedge (\forall x \in A : \mathcal{T} \leq x.\mathcal{T}_w \leq \mathcal{T}_m)$
        $\wedge v.\mathcal{T}_w = (\mathbf{max}\ x \in A : x.\mathcal{T}_w) \wedge (\forall x \in X.V - A, x.S = commit : x.\mathcal{T}_w < \mathcal{T})$
POST:   $Close(v) \wedge RET : A$
}

PROCEDURE RO-Read( V: set of versions, $\mathcal{T}$: Timestamp, var a: value)
{PRE:   $v \in V \wedge v.\mathcal{T}_w = (\mathbf{max}\ x \in X.V, x.S = commit, x.\mathcal{T}_w \leq \mathcal{T} : x.\mathcal{T}_w) \wedge v.value = A$
POST:   $a = A$
}

PROCEDURE Wait-Mess()
BEGIN
    WHILE TRUE DO
        ON reception of $ident(X) \oplus \mathcal{T}$
            V := VClose(X, $\mathcal{T}$)
            send( V)
    OD
END

PROCEDURE RO-Transaction(RS: read actions, I: data-items, t: timestamp)
VAR
    XS : set of set of versions, $\mathcal{T}_s, \mathcal{M}_{ro}$ : Timestamp
    A[data-items]: values
BEGIN
    $\mathcal{T}_s$ = f( t, time - interval); XS := $\emptyset$
    $\forall act[X] \in RS$: send( loc( X), ident(X)$\oplus\mathcal{T}_s$)
    $\forall act[X] \in RS$: receive( V); $XS := XS \cup V$
    $\mathcal{M}_{ro} := (\mathbf{min} : V \in XS : (\mathbf{max} : x \in V : x.\mathcal{T}_w))$
    $\forall V \in XS$ : RO-Read(V, $\mathcal{M}_{ro}$, A[V])
    execute calculations
END
```

Figure 30: RO-Read requests for MVTSO, one communication phase

constitutes the $\mathcal{M}_{ro}$ value. The RO-Read procedure is invoked with this $\mathcal{M}_{ro}$ values for each set of versions present in XS.

OCC-NVTI In Fig. 31, a one phase algorithm for OCC-NVTI is shown. It is assumed that a lower limit, $\mathcal{T}_a$, on the age of the acquired values can be specified. Older values are assumed to be unusable for the application. This age parameter is sent over to the participants which return committed versions which are younger than the specified age-limit and which are older than the oldest not yet commited version. On the base of the returned sets of versions, the $\mathcal{M}_{ro}$ value can be calculated and the required values can be selected locally because they are already available at the requesting processor.

The Procedure RO-Read has the same specification as the RO-Read of the former algorithm with the exception that the data-item parameter is replaced by a set of versions.

Again at every processor the Wait-Mess procedure is started which waits for messages. On reception of a message, the relevant versions are selected in V and the result set is sent back to the requester.

The RO-Transaction sends a request for the set of versions to all participants and waits for the results which consecutively are stored in the set XS. For every data-item the maximum of all C_b values of the versions is calculated. The minimum of all maxima is stored in $\mathcal{C}_m$ and constitutes the $\mathcal{M}_{ro}$ value. The RO-Read procedure is invoked with this $\mathcal{M}_{ro}$ values for each

```
PROCEDURE RO-Read( V: set of versions, C: Counter, var a: value)
{PRE:   $v \in V \wedge v.S = commit \wedge v.C_b \leq C \wedge (\forall x \in X.V, x.S = commit : x.C_b \leq v.C_b) \wedge v.value = A$
POST:   $a = A$
}

PROCEDURE Wait-Mess()
VAR
        V: set of Version
BEGIN
        WHILE TRUE DO
                ON reception of $ident(X) \oplus \mathcal{T}_a \oplus \mathcal{T}_s$
                        $V := \{v \mid v \in X.V \wedge v.S = commit \wedge v.\mathcal{T}_c < \mathcal{T}_s \wedge v.\mathcal{T}_c > \mathcal{T}_a)\}$
                        send( V )
        OD
END

PROCEDURE RO-Transaction(RS: read actions, I: data-items, t: timestamp)
VAR
        $XS$ : set of set of versions, $\mathcal{T}_s, \mathcal{T}_a : Timestamp, \mathcal{M}_{ro} : Counter$
        A[data-items]: values
BEGIN
        $\mathcal{T}_s := time \circ pid; \mathcal{T}_a = (time - interval) \circ pid; XS := \emptyset$
        $\forall act[X] \in RS$: send( loc( X), ident(X)$\oplus \mathcal{T}_a \oplus \mathcal{T}_s$)
        $\forall act[X] \in RS$: receive( V); $XS := XS \cup V$
        $\mathcal{M}_{ro} := (\min V \in XS : (\max x \in V : x.C_b))$
        $\forall V \in XS$ : RO-Read(V, $\mathcal{M}_{ro}$, A[V])
        execute calculations
END
```

Figure 31: RO-Read requests for OCC-NVTI, one communication phase

set of versions present in XS.

11.2.3 Repetitive update

MVTSO In Fig. 32 the last alternative for MVTSO is shown. Every time a Commit action is executed on a data-item X, relevant versions are sent to all destinations which want to acquire X on a regular basis. For that reason the data-item structure is extended with another field, *req*, in which all requesting processors are stored. In all processors the identifier to data-item mapping, *XS*, is stored. The Procedure *Wait-Mess*, executing on every processor, waits for messages and adds new versions to the version lists of the specified data-item replicated in *XS*.

In RO-transaction the maximum $\mathcal{T}_w$ values of the versions of a given identifier locally stored in XS are determined for all required data-item. The minimum of these values represents $\mathcal{M}_{ro}$. RO-Read is invoked on the versions of the data-items replicated in XS. The specification of RO-Read is the same as for the RO-Read in Fig. 30.

Every time a version of a requested data-item ($X.req \neq \emptyset$) is committed, the Procedure PCommit is invoked with the Commit timestamp. Pcommit invokes XClose and sends the returned versions to the requesting processors. The Function XClose returns the last committed versions which have not been sent to the requesters. At any given moment a series of already communicated versions exists. They can be identified from their $\mathcal{T}_r$ values which are equal to the $\mathcal{T}_w$ values of their immediate successors.

When a version commits, it can be a version which is the immediate successor of an

```
PROCEDURE XClose( X: set of versions, $\mathcal{T}$: Timestamp, var V: set of versions): boolean
{PRE:   $v.\mathcal{T}_m = (\mathbf{max}\ x \in X.V : x.\mathcal{T}_w) \wedge \mathcal{T} = TS \wedge TR = v.\mathcal{T}_r$
POST:   $(v.\mathcal{T}_r = TS \Rightarrow V = \emptyset \wedge RET : TRUE)$
        $\wedge(v.\mathcal{T}_r \neq TS \Rightarrow V = W \cup \{v\} \wedge RET : (v.S = commit \wedge XClose(X \setminus v, v.\mathcal{T}_w, W))$
}

PROCEDURE RO-Read( V: set of versions, $\mathcal{T}$: Timestamp, var a: value)
{PRE:   $v \in V \wedge v.\mathcal{T}_w = (\mathbf{max}\ x \in V, x.\mathcal{T}_w \leq \mathcal{T} : x.\mathcal{T}_w) \wedge v.value = A$
POST:   $a = A$
}

VAR
        $XS : ident \rightarrow data\text{-}item$

PROCEDURE PCommit( X: data-item, $\mathcal{T}$: Timestamp)
VAR
        W: set of versions
BEGIN
        $\forall dest \in X.req$: IF XClose(X, $\mathcal{T}$, W) THEN Send( dest, W) FI
END

PROCEDURE Wait-Mess()
BEGIN
        WHILE TRUE DO
                ON reception of $ident(X) \oplus W$
                        $XS(ident(X)).V := XS(ident(X)).V \cup W$
        OD
END

PROCEDURE RO-Transaction(RS: read actions, I: data-items, t: timestamp)
VAR
        $\mathcal{M}_{ro}$ : Timestamp
        A[data-items]: Values
BEGIN
        $\mathcal{M}_{ro} := (\mathbf{min}:\ act[X] \in RS : (\mathbf{max}:\ x \in XS(X).V : x.\mathcal{T}_w))$
        $\forall act[X] \in RS$ : RO-Read( XS(X).V, $\mathcal{M}_{ro}$, A[X])
        execute calculations
END
```

Figure 32: RO-Read updates for MVTSO

uncommitted version. In that case, XClose returns the empty set. When the committed version is the immediate successor to an already communicated version, then XClose returns this version and the committed immediate successors. The read timestamp of the returned versions are adjusted such that the $\mathcal{T}_r$ values of their immediate precessors are equal to their $\mathcal{T}_w$ value.

OCC-NVTI In Fig. 33 the last alternative is shown. Every time A-Commit is executed on a data-item X, relevant versions are sent to all destinations that want to acquire X on a regular basis. For that reason the data-item structure is extended with another field, *req*, in which all requesting processors are stored. In all processors the identifier to data-item mapping, *XS*, is stored. The Procedure *Wait-Mess*, executing on every processor, waits for messages and adds new versions to the version lists of the specified data-item replicated in *XS*.

In RO-transaction the maximum C_b values of the versions of a given identifier locally stored in XS are determined for all required data-item. The minimum of these values rep-

```
PROCEDURE RO-Read( V: set of versions, C: Counter, var a: value)
{PRE:  $v \in V \wedge v.C_b \leq C \wedge (\forall x \in V : x.C_b \leq v.C_b) \wedge v.value = A$
POST:  $a = A$
}

VAR
        XS : ident → data-item

PROCEDURE PCommit( X: data-item, T: Timestamp)
 VAR
        V: set of Version
BEGIN
        V := $\{v \mid v \in X.V \wedge v.S = commit\}$
        $\forall dest \in X.req$: Send( dest, V)
END

PROCEDURE Wait-Mess()
BEGIN
        WHILE TRUE DO
                ON reception of $ident(X) \oplus W$
                        XS(ident(X)).V := XS(ident(X)).V $\cup$ W
        OD
END

PROCEDURE RO-Transaction(RS: read actions, I: data-items, t: timestamp)
VAR
        $\mathcal{M}_{ro}$ : Counter
        A[data-items]: values
BEGIN
        $\mathcal{M}_{ro} := (\min X \in I : (\max x \in XS(X).V : x.C_b))$
        $\forall X \in I$ : RO-Read( XS(X).V, $\mathcal{M}_{ro}$, A[X])
        execute calculations
END
```

Figure 33: RO-Read updates for OCC-NVTI

resents $\mathcal{M}_{ro}$. RO-Read is invoked on the versions of the data-items replicated in XS. The specification of RO-Read is the same as for the RO-Read in Fig. 31.

Every time a version of a requested data-item ($X.req \neq \emptyset$) is committed, the Procedure PCommit is invoked with the Commit timestamp. Pcommit sends the returned versions to the requesting processors

12 Conclusions

An application consisting of transactions acting on a database is divided in two parts: a HRT and a SRT part. HRT- and SRT-transactions communicate via database items. The presented extensions to existing algorithms allow RO-transactions to proceed without waiting or with considerably reduced wait-times. The concept of transfer serializability is formulated to increase the number of interleavings between HRT and SRT transactions.

13 Acknowledgements

K. Vidyanshar has pointed out a mistake in the order definition proposed in an earlier version of this document. This work has benefited from discussions with P.T.A. Thijssen. A.Engel

and A.Glim have contributed to the specifications of the ACTIONS presented here.

References

[AL81] T. Anderson and P.A. Lee. *Fault Tolerance, Principles and Practice.* Prentice-Hall, Inc., 1981.

[BBG89] C. Beeri, P.A. Bernstein, and N. Goodman. A model for concurrency in nested transaction systems. *Journal of the ACM*, 36(2):230–269, 1989.

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

[GMA87] H. Garcia-Molina and R.K. Abbott. Reliable Distributed Database management. *Proc. of IEEE*, 75(5):601–620, 1987.

[Gra92] M.H. Graham. Issues in Real-Time Data Management. *Journal of Real-Time Systems*, 4:185–202, 1992.

[HSRT91] J. Huang, J.A. Stankovic, K. Ramamritham, and D. Towsley. Experimental evaluation of real-time optimistic concurrency control schemes. In *Proc of Very Large Databases*, 1991.

[Kna87] E. Knapp. Deadlock detection in distributed databases. *ACM Computing Surveys*, 19(4), 1987.

[KR81] H.T. Kung and J.T. Robinson. On optimistic methods for concurrency control in distributed database systems. *ACM Transactions on database systems*, 6(2):213–226, 1981.

[LS93] J. Lee and S.H. Son. Using Dynamic Adjustment of Serialization Order for Real-Time Database Systems. In *Proceedings of 14th Real-Time Systems Symposium*, pages 66–75, Raleigh-Durham, December 1993. IEEE.

[Mos85] J.E.B. Moss. *Nested Transactions: an approach to reliable distributed computing.* PhD thesis, MIT, Cambridge, Mass., 1985.

[Mos87] J.E.B. Moss. Nested transactions: An introduction. In Bhargava, editor, *Concurrency control and reliability in distributed systems*, 1987.

[Pap86] C.H. Papadimitriou. *The theory of Database Concurrency Control.* Computer Science Press, 1986.

[Ram93] K. Ramamritham. Real-Time Databases. *Distributed and Parallel Databases*, 1993(1):199–226, 1993.

[Ree83] D.P. Reed. Implementing methods for concurrency control in distributed database systems. *ACM Transactions on Computer Systems*, 1(1):3–23, 1983.

[Sch81] N. Schlageter. Optimisitic methods for concurrency control in distributed database systems. In *Proceedings of Very Large Databases*, pages 125–130, 1981.

[SMB85] S.K. Shrivastava, L.V. Mancini, and B.Randell. On the Duality of Process and Object Model. In S.K. Shrivastava, editor, *Reliable Computer Systems*, pages 19–37. Springer Verlag, 1985.

[Vid85] K. Vidyasankar. A simple characterization of database serializability. In *5th conf. on Foundations of Software Technology and Theoretical Computer Science.* Springer Verlag, 1985.

[Vid91] K. Vidyasankar. Unified Theory of Database serializability. *Fundamenta Informatica*, 14:147–183, 1991.

[Wei87] W.E. Weihl. Distributed version management for read-only actions. *IEEE Transactions on Computer Systems*, 13(1):55–64, 1987.

93/31	W. Körver	Derivation of delay insensitive and speed independent CMOS circuits, using directed commands and production rule sets, p. 40.
93/32	H. ten Eikelder and H. van Geldrop	On the Correctness of some Algorithms to generate Finite Automata for Regular Expressions, p. 17.
93/33	L. Loyens and J. Moonen	ILIAS, a sequential language for parallel matrix computations, p. 20.
93/34	J.C.M. Baeten and J.A. Bergstra	Real Time Process Algebra with Infinitesimals, p.39.
93/35	W. Ferrer and P. Severi	Abstract Reduction and Topology, p. 28.
93/36	J.C.M. Baeten and J.A. Bergstra	Non Interleaving Process Algebra, p. 17.
93/37	J. Brunekreef J-P. Katoen R. Koymans S. Mauw	Design and Analysis of Dynamic Leader Election Protocols in Broadcast Networks, p. 73.
93/38	C. Verhoef	A general conservative extension theorem in process algebra, p. 17.
93/39	W.P.M. Nuijten E.H.L. Aarts D.A.A. van Erp Taalman Kip K.M. van Hee	Job Shop Scheduling by Constraint Satisfaction, p. 22.
93/40	P.D.V. van der Stok M.M.M.P.J. Claessen D. Alstein	A Hierarchical Membership Protocol for Synchronous Distributed Systems, p. 43.
93/41	A. Bijlsma	Temporal operators viewed as predicate transformers, p. 11.
93/42	P.M.P. Rambags	Automatic Verification of Regular Protocols in P/T Nets, p. 23.
93/43	B.W. Watson	A taxomomy of finite automata construction algorithms, p. 87.
93/44	B.W. Watson	A taxonomy of finite automata minimization algorithms, p. 23.
93/45	E.J. Luit J.M.M. Martin	A precise clock synchronization protocol,p.
93/46	T. Kloks D. Kratsch J. Spinrad	Treewidth and Patwidth of Cocomparability graphs of Bounded Dimension, p. 14.
93/47	W. v.d. Aalst P. De Bra G.J. Houben Y. Kornatzky	Browsing Semantics in the "Tower" Model, p. 19.
93/48	R. Gerth	Verifying Sequentially Consistent Memory using Interface Refinement, p. 20.
94/01	P. America M. van der Kammen R.P. Nederpelt O.S. van Roosmalen H.C.M. de Swart	The object-oriented paradigm, p. 28.
94/02	F. Kamareddine R.P. Nederpelt	Canonical typing and Π-conversion, p. 51.
94/03	L.B. Hartman K.M. van Hee	Application of Marcov Decision Processe to Search Problems, p. 21.
94/04	J.C.M. Baeten J.A. Bergstra	Graph Isomorphism Models for Non Interleaving Process Algebra, p. 18.
94/05	P. Zhou J. Hooman	Formal Specification and Compositional Verification of an Atomic Broadcast Protocol, p. 22.
94/06	T. Basten T. Kunz J. Black M. Coffin D. Taylor	Time and the Order of Abstract Events in Distributed Computations, p. 29.
94/07	K.R. Apt R. Bol	Logic Programming and Negation: A Survey, p. 62.
94/08	O.S. van Roosmalen	A Hierarchical Diagrammatic Representation of Class Structure, p. 22.
94/09	J.C.M. Baeten J.A. Bergstra	Process Algebra with Partial Choice, p. 16.

94/10	T. verhoeff	The testing Paradigm Applied to Network Structure. p. 31.
94/11	J. Peleska C. Huizing C. Petersohn	A Comparison of Ward & Mellor's Transformation Schema with State- & Activitycharts, p. 30.
94/12	T. Kloks D. Kratsch H. Müller	Dominoes, p. 14.
94/13	R. Seljée	A New Method for Integrity Constraint checking in Deductive Databases, p. 34.
94/14	W. Peremans	Ups and Downs of Type Theory, p. 9.
94/15	R.J.M. Vaessens E.H.L. Aarts J.K. Lenstra	Job Shop Scheduling by Local Search, p. 21.
94/16	R.C. Backhouse H. Doornbos	Mathematical Induction Made Calculational, p. 36.
94/17	S. Mauw M.A. Reniers	An Algebraic Semantics of Basic Message Sequence Charts, p. 9.
94/18	F. Kamareddine R. Nederpelt	Refining Reduction in the Lambda Calculus, p. 15.
94/19	B.W. Watson	The performance of single-keyword and multiple-keyword pattern matching algorithms, p. 46.
94/20	R. Bloo F. Kamareddine R. Nederpelt	Beyond β-Reduction in Church's $\lambda\rightarrow$, p. 22.
94/21	B.W. Watson	An introduction to the Fire engine: A C++ toolkit for Finite automata and Regular Expressions.
94/22	B.W. Watson	The design and implementation of the FIRE engine: A C++ toolkit for Finite automata and regular Expressions.
94/23	S. Mauw and M.A. Reniers	An algebraic semantics of Message Sequence Charts, p. 43.
94/24	D. Dams O. Grumberg R. Gerth	Abstract Interpretation of Reactive Systems: Abstractions Preserving $\forall$CTL*, $\exists$CTL* and CTL*, p. 28.
94/25	T. Kloks	$K_{1,3}$-free and W_4-free graphs, p. 10.
94/26	R.R. Hoogerwoord	On the foundations of functional programming: a programmer's point of view, p. 54.
94/27	S. Mauw and H. Mulder	Regularity of BPA-Systems is Decidable, p. 14.
94/28	C.W.A.M. van Overveld M. Verhoeven	Stars or Stripes: a comparative study of finite and transfinite techniques for surface modelling, p. 20.
94/29	J. Hooman	Correctness of Real Time Systems by Construction, p. 22.
94/30	J.C.M. Baeten J.A. Bergstra Gh. Ştefanescu	Process Algebra with Feedback, p. 22.
94/31	B.W. Watson R.E. Watson	A Boyer-Moore type algorithm for regular expression pattern matching, p. 22.
94/32	J.J. Vereijken	Fischer's Protocol in Timed Process Algebra, p. 38.
94/33	T. Laan	A formalization of the Ramified Type Theory, p.40.
94/34	R. Bloo F. Kamareddine R. Nederpelt	The Barendregt Cube with Definitions and Generalised Reduction, p. 37.
94/35	J.C.M. Baeten S. Mauw	Delayed choice: an operator for joining Message Sequence Charts, p. 15.
94/36	F. Kamareddine R. Nederpelt	Canonical typing and Π-conversion in the Barendregt Cube, p. 19.
94/37	T. Basten R. Bol M. Voorhoeve	Simulating and Analyzing Railway Interlockings in ExSpect, p. 30.
94/38	A. Bijlsma C.S. Scholten	Point-free substitution, p. 10.

94/39	A. Blokhuis T. Kloks	On the equivalence covering number of splitgraphs, p. 4.
94/40	D. Alstein	Distributed Consensus and Hard Real-Time Systems, p. 34.
94/41	T. Kloks D. Kratsch	Computing a perfect edge without vertex elimination ordering of a chordal bipartite graph, p. 6.
94/42	J. Engelfriet J.J. Vereijken	Concatenation of Graphs, p. 7.
94/43	R.C. Backhouse M. Bijsterveld	Category Theory as Coherently Constructive Lattice Theory: An Illustration, p. 35.
94/44	E. Brinksma R. Gerth W. Janssen S. Katz M. Poel C. Rump J. Davies S. Graf B. Jonsson G. Lowe A. Pnueli J. Zwiers	Verifying Sequentially Consistent Memory, p. 160
94/45	G.J. Houben	Tutorial voor de ExSpect-bibliotheek voor "Administratieve Logistiek", p. 43.
94/46	R. Bloo F. Kamareddine R. Nederpelt	The λ-cube with classes of terms modulo conversion, p. 16.
94/47	R. Bloo F. Kamareddine R. Nederpelt	On Π-conversion in Type Theory, p. 12.
94/48	Mathematics of Program Construction Group	Fixed-Point Calculus, p. 11.
94/49	J.C.M. Baeten J.A. Bergstra	Process Algebra with Propositional Signals, p. 25.
94/50	H. Geuvers	A short and flexible proof of Strong Normalazation for the Calculus of Constructions, p. 27.
94/51	T. Kloks D. Kratsch H. Müller	Listing simplicial vertices and recognizing diamond-free graphs, p. 4.
94/52	W. Penczek R. Kuiper	Traces and Logic, p. 81
94/53	R. Gerth R. Kuiper D. Peled W. Penczek	A Partial Order Approach to Branching Time Logic Model Checking, p. 20.
95/01	J.J. Lukkien	The Construction of a small CommunicationLibrary, p.16.
95/02	M. Bezem R. Bol J.F. Groote	Formalizing Process Algebraic Verifications in the Calculus of Constructions, p.49.
95/03	J.C.M. Baeten C. Verhoef	Concrete process algebra, p. 134.
95/04	J. Hidders	An Isotopic Invariant for Planar Drawings of Connected Planar Graphs, p. 9.
95/05	P. Severi	A Type Inference Algorithm for Pure Type Systems, p.20.
95/06	T.W.M. Vossen M.G.A. Verhoeven H.M.M. ten Eikelder E.H.L. Aarts	A Quantitative Analysis of Iterated Local Search, p.23.
95/07	G.A.M. de Bruyn O.S. van Roosmalen	Drawing Execution Graphs by Parsing, p. 10.
95/08	R. Bloo	Preservation of Strong Normalisation for Explicit Substitution, p. 12.
95/09	J.C.M. Baeten J.A. Bergstra	Discrete Time Process Algebra, p. 20
95/10	R.C. Backhouse R. Verhoeven O. Weber	Math∫pad: A System for On-Line Prepararation of Mathematical Documents, p. 15

95/11	R. Seljée	Deductive Database Systems and integrity constraint checking, p. 36.
95/12	S. Mauw and M. Reniers	Empty Interworkings and Refinement Semantics of Interworkings Revised, p. 19.
95/13	B.W. Watson and G. Zwaan	A taxonomy of sublinear multiple keyword pattern matching algorithms, p. 26.
95/14	A. Ponse, C. Verhoef, S.F.M. Vlijmen (eds.)	De proceedings: ACP'95, p.
95/15	P. Niebert and W. Penczek	On the Connection of Partial Order Logics and Partial Order Reduction Methods, p. 12.
95/16	D. Dams, O. Grumberg, R. Gerth	Abstract Interpretation of Reactive Systems: Preservation of CTL*, p. 27.
95/17	S. Mauw and E.A. van der Meulen	Specification of tools for Message Sequence Charts, p. 36.
95/18	F. Kamareddine and T. Laan	A Reflection on Russell's Ramified Types and Kripke's Hierarchy of Truths, p. 14.
95/19	J.C.M. Baeten and J.A. Bergstra	Discrete Time Process Algebra with Abstraction, p. 15.
95/20	F. van Raamsdonk and P. Severi	On Normalisation, p. 33.
95/21	A. van Deursen	Axiomatizing Early and Late Input by Variable Elimination, p. 44.
95/22	B. Arnold, A. v. Deursen, M. Res	An Algebraic Specification of a Language for Describing Financial Products, p. 11.
95/23	W.M.P. van der Aalst	Petri net based scheduling, p. 20.
95/24	F.P.M. Dignum, W.P.M. Nuijten, L.M.A. Janssen	Solving a Time Tabling Problem by Constraint Satisfaction, p. 14.
95/25	L. Feijs	Synchronous Sequence Charts In Action, p. 36.
95/26	W.M.P. van der Aalst	A Class of Petri nets for modeling and analyzing business processes, p. 24.
95/27	P.D.V. van der Stok, J. van der Wal	Proceedings of the Real-Time Database Workshop, p. 106.
95/28	W. Fokkink, C. Verhoef	A Conservative Look at term Deduction Systems with Variable Binding, p. 29.
95/29	H. Jurjus	On Nesting of a Nonmonotonic Conditional, p. 14
95/30	J. Hidders, C. Hoskens, J. Paredaens	The Formal Model of a Pattern Browsing Technique, p.24.
95/31	P. Kelb, D. Dams and R. Gerth	Practical Symbolic Model Checking of the full μ-calculus using Compositional Abstractions, p. 17.
95/32	W.M.P. van der Aalst	Handboek simulatie, p. 51.
95/33	J. Engelfriet and JJ. Vereijken	Context-Free Graph Grammars and Concatenation of Graphs, p. 35.
95/34	J. Zwanenburg	Record concatenation with intersection types, p. 46.
95/35	T. Basten and M. Voorhoeve	An algebraic semantics for hierarchical P/T Nets, p. 32.
96/01	M. Voorhoeve and T. Basten	Process Algebra with Autonomous Actions, p. 12.
96/02	P. de Bra and A. Aerts	Multi-User Publishing in the Web: DreSS, A Document Repository Service Station, p. 12
96/03	W.M.P. van der Aalst	Parallel Computation of Reachable Dead States in a Free-choice Petri Net, p. 26.
96/04	S. Mauw	Example specifications in phi-SDL.
96/05	T. Basten and W.M.P. v.d. Aalst	A Process-Algebraic Approach to Life-Cycle Inheritance Inheritance = Encapsulation + Abstraction, p. 15.
96/06	W.M.P. van der Aalst and T. Basten	Life-Cycle Inheritance A Petri-Net-Based Approach, p. 18.
96/07	M. Voorhoeve	Structural Petri Net Equivalence, p. 16.
96/08	A.T.M. Aerts, P.M.E. De Bra, J.T. de Munk	OODB Support for WWW Applications: Disclosing the internal structure of Hyperdocuments, p. 14.
96/09	F. Dignum, H. Weigand, E. Verharen	A Formal Specification of Deadlines using Dynamic Deontic Logic, p. 18.
96/10	R. Bloo, H. Geuvers	Explicit Substitution: on the Edge of Strong Normalisation, p. 13.
96/11	T. Laan	AUTOMATH and Pure Type Systems, p. 30.
96/12	F. Kamareddine and T. Laan	A Correspondence between Nuprl and the Ramified Theory of Types, p. 12.
96/13	T. Borghuis	Priorean Tense Logics in Modal Pure Type Systems, p. 61

96/14	S.H.J. Bos and M.A. Reniers	The I^2C-bus in Discrete-Time Process Algebra, p. 25.
96/15	M.A. Reniers and J.J. Vereijken	Completeness in Discrete-Time Process Algebra, p. 139.
96/16	P. Hoogendijk and O. de Moor	What is a data type?, p. 29.
96/17	E. Boiten and P. Hoogendijk	Nested collections and polytypism, p. 11.