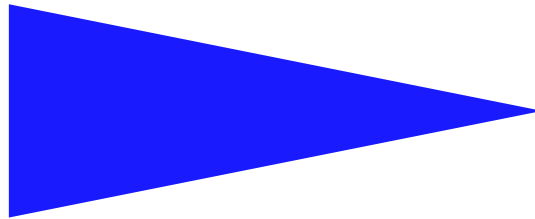




PUBLICATION
INTERNE
N° 1917



**ON THE CONSISTENCY CONDITIONS
OF TRANSACTIONAL MEMORIES**

DAMIEN IMBS JOSÉ R. G. DE MENDÍVIL MICHEL RAYNAL

On the Consistency Conditions of Transactional Memories

Damien Imbs* José R. G. de Mendivil** Michel Raynal***

Systèmes communicants
Projet ASAP

Publication interne n° 1917 — Décembre 2008 — 23 pages

Abstract: The aim of a Software Transactional Memory (STM) is to discharge the programmers from the management of synchronization in multiprocess programs that access concurrent objects. To that end, a STM system provides the programmer with the concept of a *transaction*: each sequential process is decomposed into transactions, where a transaction encapsulates a piece of sequential code accessing concurrent objects. A transaction contains no explicit synchronization statement and appears as if it has been executed atomically. Due to the underlying concurrency management, a transaction commits or aborts. Up to now, few papers focused on the definition of consistency conditions suited to STM systems. One of them has recently proposed the opacity consistency condition. Opacity involves all the transactions (i.e., the committed plus the aborted transactions). It requires that (1) until it aborts (if ever it does) a transaction sees a consistent global state of the concurrent objects, and (2) the execution is linearizable (i.e., it could have been produced by a sequential execution -of the same transactions- that respects the real time order on the non-concurrent transactions).

This paper is on consistency conditions for transactional memories. It first presents a framework that allows defining a space of consistency conditions whose extreme endpoints are serializability and opacity. It then extracts from this framework a new consistency condition that we call *virtual world* consistency. This condition ensures that (1) each transaction (committed or aborted) reads values from a consistent global state, (2) the consistent global states read by committed transactions are mutually consistent, but (3) the consistent global states read by aborted transactions are not required to be mutually consistent. Interestingly enough, this consistency condition can benefit lots of STM applications as, from its local point of view, a transaction cannot differentiate it from opacity. Finally, the paper presents and proves correct a STM algorithm that implements the virtual world consistency condition. Interestingly, this algorithm distinguishes the serialization date of a transaction from its commit date (thereby allowing more transactions to commit).

Key-words: Atomic operation, Concurrency control, Concurrent programming, Consistent global state, Consistency condition, Incremental snapshot, Linearizability, Lock, Opacity, Serializability, Shared object, Software transactional memory, Transaction.

(Résumé : tsyp)

* IRISA, Université de Rennes 1, Campus de Beaulieu, 35042 Rennes Cedex, France damien.imbs@irisa.fr

** Dept. Informática, Universidad Pública de Navarra, Pamplona, Spain, mendivil@unavarra.es

*** IRISA, Université de Rennes 1, Campus de Beaulieu, 35042 Rennes Cedex, France, raynal@irisa.fr



Sur les critères de cohérence des mémoires transactionnelles

Résumé : Ce rapport commence par analyser les critères de cohérence proposés à ce jour pour les mémoires transactionnelles. A partir d'une vue synthétique de ces derniers, il propose un nouveau critère appelé *monde virtuel*. Plus faible que l'opacité (un critère très intéressant proposé dernièrement), le critère *monde virtuel* en garde l'esprit, à savoir qu' indépendamment du fait qu'une transaction soit validée ou avortée, celle-ci ne lit jamais de valeurs incohérentes. Un protocole qui garantit ce critère est ensuite présenté et prouvé formellement correct.

Mots clés : Atomicité, Contrôle de la concurrence, Etat global cohérent, Mémoire transactionnelle, Object partagé, Opacité, Transaction, Validation, Verrou.

1 Introduction

Software transactional memory The concept of *Software Transactional Memory* (STM), originally proposed in [33], originates from the observation that programmers were missing something for the applications made up of concurrent processes that access shared data structures (base objects). Roughly speaking, the “only” tool they were proposed to solve their synchronization problems is a set of locks that allow the processes to prevent conflicting accesses to shared objects. But, due to the difficulty to manage them, locks are not a panacea. If a single lock controls a large set of data, it reduces drastically the parallelism, while many locks used to control fine grain data are difficult to master and error-prone. On another side, the recent advances in technology (e.g., more particularly in multicore architectures) have given rise to a new momentum to practical and theoretical research in concurrency and synchronization that constitutes a large application domain strongly related to STM systems [3, 12, 16].

The STM approach is a middleware approach that allows the programmers to write their applications in terms of *transaction*-based processes¹. Fundamentally, it provides a “new programming construct that offers a higher-level abstraction for writing parallel programs” [25]. More precisely, an application program is made up of processes (or automata), each process being defined as (or decomposed into) a sequence of transactions, where a transaction is a sequential piece of code that, while accessing any number of shared base objects in a concurrency context, appears as being executed atomically. The code of a transaction is not known in advance and a priori no two transactions have the same code. The job of the programmer is only to define the units of computation that are the transactions. He does not have to worry about the fact that the base objects can be concurrently accessed by transactions. Except when he defines the beginning and the end of a transaction (that are the only synchronization points known by a process), the programmer is not concerned by synchronization. It is the job of the STM system to ensure that transactions execute as if they were atomic (in addition to transactions, a process can contain additional code, but we do not consider such a possibility in this paper). At the programming level, the transactions are a way to structure each process into a sequence of atomic processing units, each transaction being defined as a synchronization-free sequential code.

A STM system is a software device whose inputs are process-generated transactions, that produces a corresponding run that, to be meaningful, has to satisfy some properties. Each set of properties defines a particular STM system. Of course, a solution in which a single transaction at a time would execute, trivially implements transaction atomicity but would be irrelevant from an efficiency point of view. So, a STM system has to do “its best” to execute as many transactions per time unit as possible. Similarly to a scheduler, a STM system is an on-line algorithm that does not know the future. If the STM is not trivial (i.e., it allows several transactions that access the same objects in a conflicting manner to run concurrently), this intrinsic limitation has a price, namely, it can direct transactions to abort in order to guarantee both transaction atomicity and object consistency. From a programming point of view, an aborted transaction has no effect (it is up to the process that issued an aborted transaction to re-issue it or not; usually, a transaction that is restarted is considered as a new transaction).

STM consistency The major part of the STM systems that have been designed so far are mainly efficiency-oriented. They strive to improve the number of transactions that are committed per time unit. As far as the correctness condition they ensure is concerned, they implement serializability [7, 27], or variants of it (e.g., [5, 17, 32]). Very little effort has been devoted to precisely define the properties a STM implementation has to satisfy.

It has recently been suggested that a transaction, whatever its fate (commit or abort), should not obtain values of the objects it reads from an inconsistent global state of the shared memory [11]. This consis-

¹Differently, in classical transactional systems (such as database), there is no structuring notion of process.

tency condition has been formalized and investigated in [14], where it has been given the name *opacity*. Intuitively, this means that the values read by any transaction have to be mutually consistent. (As a side effect, a transaction that aborts has to be “reduced” by the STM system to a consistent read prefix.) Several algorithms that have been proposed do implement the opacity property [8, 11, 19, 21, 30, 34] (some of them have been designed before the property was explicitly formulated, while others considered it as their explicit consistency condition).

Content of the paper Serializability and opacity can be seen as two extreme consistency conditions. Serializability (mainly used in database) involves only the committed transactions and states that an execution is correct if it could have been produced by a sequential execution of the committed transactions. Strict serializability is a strengthening that adds the requirement that the “witness” sequential execution should respect the real time order of non-concurrent transactions.

In a database system, a transaction is usually the result of a query formulated in a specific language. Differently, a transaction T in a STM system can be any (correct) piece of code. Correct means that, if the objects in the shared memory are mutually consistent and T is executed in a concurrency-free context, it behaves correctly, i.e., T provides the invoking process with correct values, and (if any) the modifications of the base objects issued by T are in agreement with their specification. If, in a concurrency context, a transaction is allowed to read object values that are mutually inconsistent, albeit its code is correct, it can be directed to behave arbitrarily before being aborted (e.g., entering an infinite loop or dividing by zero.) Moreover, it can be difficult (or even impossible in some cases) to distinguish an arbitrary behavior from a correct behavior of a transaction.

Opacity solves this problem by preventing any transaction from reading an inconsistent state of the base objects. If it is about to read an object whose current value would make inconsistent its previous reads, a transaction is aborted. Reducing each aborted transaction to such a consistent read prefix, opacity involves all the transactions (committed and reduced aborted transactions) and requires that the corresponding execution be equivalent to a sequential execution that respects the real time order on non-concurrent transactions.

The paper first introduces a consistency condition space that lies between serializability and opacity, explores and investigates it, and shows that there are consistency conditions that, while weaker than opacity, can benefit STM systems. Among these consistency conditions, *virtual world* consistency seems particularly interesting. As opacity, this condition involves all the transactions (all the committed transactions and the reduced aborted transactions). Intuitively, it requires that (1) all the committed transactions be serializable, and (2) each (reduced) aborted transaction T reads values that are mutually consistent when considering its causal past only. It is important to see that while all the committed transactions have the same witness sequential execution, each aborted transaction has its own witness sequential execution that involves only its past. Two aborted transactions can have different witnesses, both providing them with consistent values, but may be incompatible with respect to each other, hence the name “virtual world”.

Then, replacing serializability of committed transactions by strict serializability, and requiring the witness sequential execution of each aborted transaction to respect or not real time defines a family of consistency conditions.

The paper presents then an algorithm that constructs a virtual world consistent STM system. This algorithm never aborts a write-only transaction, and (differently from the algorithms that ensure opacity) makes a distinction between the serialization time of a transaction and its commit time. This feature can open the way to a new family of STM algorithms.

Roadmap The paper is made up of 6 sections. Section 2 presents the computation model and introduces base definitions. Section 3 introduces a general framework from which is defined a space of consistency conditions. Virtual world consistency is one of the conditions that lies in this space. Then, Section 4 presents an algorithm implementing a virtual world consistent STM system. This algorithm is proved correct in Section 5. Finally, Section 6 concludes the paper.

2 Computation model and base definitions

2.1 Processes and base objects

From an application point of view, a system is made up of a set of n processes p_1, \dots, p_n , plus a set of base concurrent objects accessed by atomic read and write operations. There is no assumption on the respective speed of processes, except they are neither zero, nor infinite: the processes are *asynchronous*.

2.2 Transactions and base events

Transaction A transaction is a piece of code that is produced on-line by a sequential process (automaton), that is assumed to be executed atomically (commit) or not at all (abort). This means that (1) the transactions issued by a process are totally ordered, and (2) the designer of a transaction has not to worry about the management of the base objects accessed by the transaction. Differently from a committed transaction, an aborted transaction has no effect on the shared objects. A transaction can read or write any base object. Such a read or write access is atomic. The set of the objects read by a transaction defines its *read set*. Similarly the set of objects it writes defines its *write set*. A transaction that does not write base objects is a *read-only* transaction, otherwise it is an *update* transaction. A transaction that issues only write operations is a *write-only* transaction.

As in [7], we consider that the behavior of a transaction T can be decomposed in three sequential steps²: it first reads data objects, then does local computations and finally writes new values in some objects, which means that a transaction can be seen as a software `read_modify_write()` operation that is dynamically defined by a process³. The read set is defined incrementally, which means that a transaction reads the objects of its read set asynchronously one after the other (between two consecutive reads, the transaction can issue local computations that take arbitrary, but finite, durations). We say that the transaction T computes an *incremental snapshot*⁴. This snapshot has to be *consistent* which means that there is a time frame in which these values have co-existed (as we will see later, different consistency conditions consider different time frame notions). If it is about to read a new object whose current value would make inconsistent its current incremental snapshot, the transaction T is directed to abort. If it is not aborted during its read phase, T issues local computations. Finally, if T is an update transaction, and its write operations can be issued in such a way that T appears as being executed atomically, the objects of its write set are updated and T commits; otherwise, T is aborted. So, each aborted transaction is reduced to a read prefix. When, at the model level in the following, we speak about an aborted transaction, we implicitly refer to such a prefix. Independently of consistency reasons, a transaction T can also be aborted by the process that issued

²This model is for reasoning, understand and state properties on STM systems. It only requires that everything appears as described in the model. It does not preclude an implementation where a transaction writes some objects before reading other objects. In that case, a transaction that aborts has to undo its previous writes.

³Different `read_modify_write()` operations are provided by some processors. Classical examples of such operations provided by hardware are the instructions `test&set()`, `fetch&increment()`, and `compare&swap()`. Their read set is equal to their write set, and contain a single atomic register. Moreover, their internal computation is defined once for all.

⁴The incremental approach to compute a snapshot reads asynchronously (separately) one object after the other. Differently, in [1, 4, 20], the whole set of the base objects to be atomically read is globally defined at the time of the snapshot invocation.

it. (From our point of view, namely the definition of *consistency conditions* for STM systems, we consider that such aborts include the case where transactions are aborted in order to improve the global efficiency⁵.)

Events at the shared memory level Each transaction generates events defined as follows.

- **Begin and end events.** The event denoted B_T is associated with the beginning of the transaction T , while the event E_T is associated with its termination. E_T can be of two types, namely A_T and C_T , where A_T is the event “abort of T ”, while C_T is the event “commit of T ”.
- **Read events.** The event denoted $r_T(X)v$ is associated with the atomic read of X (from the shared memory) issued by the transaction T . The value v denotes the value returned by the read. If the value v , or T , is irrelevant $r_T(X)v$ is abbreviated $r_T(X)$, or $r(X)v$ or $r(X)$. The notation $r_T(X)v \in T$, or $r(X)v \in T$, or $r(X) \in T$, is used to express that $r_T(X)v$ is an event of T .
- **Write events.** The event denoted $w_T(X)v$ is associated with the atomic write of the value v in the shared object X (in the shared memory). If the value v is irrelevant $w_T(X)v$ is abbreviated $w_T(X)$. Without loss of generality we assume that no two writes on the same object X write the same value. We also assume that all the objects are initially written by a fictitious transaction. Similarly to the previous item, the notation $w_T(X)v \in T$, or $w(X)v \in T$, or $w(X) \in T$, is used to express that $w_T(X)v$ is an event of T .

At the shared memory level, only the events such as B_T , E_T , $r_T(X)v$ and $w_T(X)v$ are perceived. Let H be the set of all these events. Moreover, as $r_T(X)v$ and $w_T(X)v$ correspond to the execution of base atomic operations, the set of all the begin, end, read and write events can be totally ordered. This total order, denoted $\widehat{H} = (H, <_H)$, is called a *shared memory history*.

2.3 Execution histories

Transaction history The execution of a set of transactions is represented by a partial order $\widehat{PO} = (PO, \rightarrow_{PO})$ that expresses a structural property of the execution of these transactions capturing the order of these transactions as issued by the processes and in agreement with the values they have read. More formally, we have:

- PO is the set of transactions, and
- $T1 \rightarrow_{PO} T2$ (we say “ $T1$ precedes $T2$ ”) if:
 1. (Process order.) Both $T1$ and $T2$ have been issued by the same process, and $T1$ is a committed transaction that has been issued before $T2$.
 2. (Read_from order.) $\exists w_{T1}(X)v \wedge \exists r_{T2}(X)v$. (There is an object X whose value written by $T1$ has been read by $T2$.)
 3. (Transitivity.) $\exists T : (T1 \rightarrow_{PO} T) \wedge (T \rightarrow_{PO} T2)$.

Remark When we look at the partial order \widehat{PO} , it is important to notice that, while all the committed transactions issued by a process are totally ordered, there is no precedence edge that originates from an aborted transaction. For the committed transactions issued by a process, this expresses the fact that those have been sequentially issued by that process and are possibly causally related. Roughly speaking, this total order defines what that process “really did”. Differently, whatever the values read by an aborted transaction

⁵This is the case for example in the system TL2 [11] where a transaction can be sacrificed (aborted) to increase the number of transactions that are committed per time unit. This occurs when a transaction tries to lock an object that is already locked.

(a priori those can be mutually consistent or not), those values do not have to “causally” impact the future in a systematic way (except if a process voluntarily takes them into account in its next transaction).

As we can see, an important difference between classical (e.g., database) transactions and STM transactions lies in the fact that in a STM the transactions are issued by processes. (In a database, there is no notion of process that relates transactions.) Of course, in a STM system, it could be possible to ask a process to indicate which of its transactions are process-order related. This possibility would add flexibility (and could be relevant for some applications) but does not change fundamentally the process-based model previously introduced.

Independent transactions and sequential execution Given a partial order $\widehat{PO} = (PO, \rightarrow_{PO})$ that models a transaction execution, two transactions T_1 and T_2 are *independent* (or concurrent) if neither is ordered before the other: $\neg(T_1 \rightarrow_{PO} T_2) \wedge \neg(T_2 \rightarrow_{PO} T_1)$. An execution such that \rightarrow_{PO} is a total order, is a *sequential* execution.

Committed transaction history A *committed transaction history* (in short c-history) is a partial order \widehat{CH} as defined above where the set of transactions (denoted CH) is made up of all the committed transactions. Moreover, \rightarrow_{PO} is then denoted \rightarrow_{CH} .

An example of such a partial order is described in Figure 1, where a committed transaction is depicted by a big black dot. The “time line” of each process is indicated with a slim long horizontal arrow. The precedence edges of the \rightarrow_{PO} relation are indicated with black arrows. Assuming that the transactions access the base objects x, y and z , some read-from edges are indicated by labeled arrows where the label indicates the object written and read respectively by the endpoint transactions (the corresponding object values are not represented). Transitivity edges are not represented.

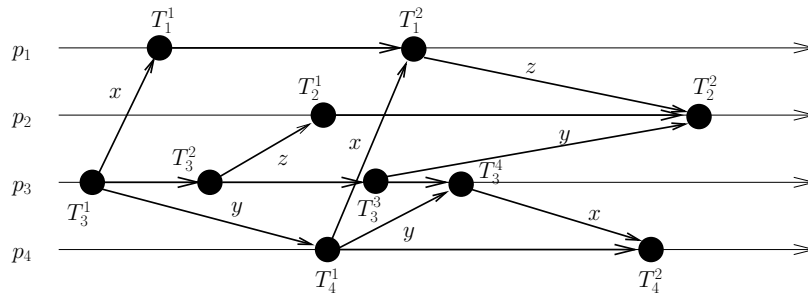


Figure 1: A partial order $\widehat{CH} = (CH, \rightarrow_{CH})$ (only committed transactions)

Complete transaction history A *complete transaction history* (in short ca-history) is a partial order \widehat{CAH} as defined above where the set of transactions (denoted CAH) is made up of all the committed or aborted transactions. The order relation \rightarrow_{PO} is denoted \rightarrow_{CAH} . Let us observe that $\rightarrow_{CH} \subseteq \rightarrow_{CAH}$.

Let T be an aborted transaction. If T reads, we have directed edges $T' \rightarrow_{CAH} T$ where T' is a committed transaction. Moreover, it follows from (1) the fact that an aborted transaction T does not write the shared memory, and (2) the definition of the process order relation, that there is no outgoing edge from an aborted transaction T .

Figure 2 describes a \widehat{CAH} partial order in which the aborted transactions are depicted with squares (those are denoted T_2', T_3' and T_4'). When considering T_2' , the figure shows that it reads two values one produced by T_1^1 , the other by T_3^4 . The arrow from T_1^1 to T_2' is a process order edge (and there is no process edge from T_2' to T_2^2).

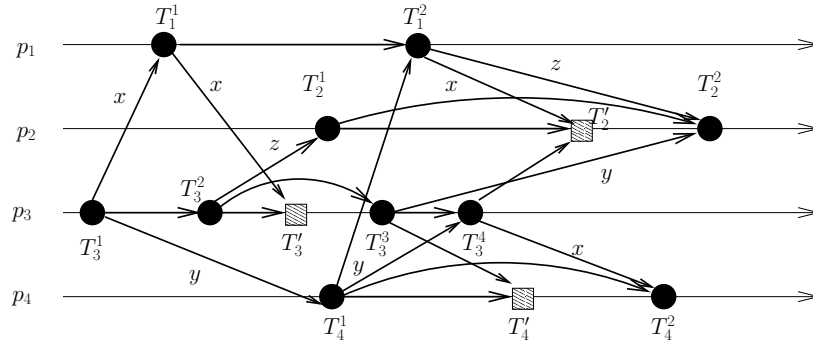


Figure 2: A partial order $\widehat{CAH} = (CAH, \rightarrow_{CAH})$ (committed and aborted transactions)

2.4 Additional base definitions

Real time order Let \rightarrow_{RT} be the *real time* relation defined as follows: $T1 \rightarrow_{RT} T2$ if E_{T1} occurs before B_{T2} ($E_{T1} <_H B_{T2}$). This relation (defined on the whole set of transactions, or only the committed transactions) is a partial order. In the particular case where it is a total order, we say that we have a real time-complying sequential execution.

Considering that the space/time diagrams depicted in the previous Figures 1 and 2 are real time diagrams, we see that $T_1^1 \rightarrow_{RT} T_3^4$, while the executions of T_2^1 and T_4^1 overlap in real time.

Linear extension A linear extension $\widehat{S} = (S, \rightarrow_S)$ of a partial order $\widehat{PO} = (PO, \rightarrow_{PO})$ is a topological sort of this partial order, i.e.,

- $S = PO$ (same elements),
- \rightarrow_S is a total order, and
- $(T1 \rightarrow_{PO} T2) \Rightarrow (T1 \rightarrow_S T2)$ (we say that \rightarrow_S respects \rightarrow_{PO}).

As an example the sequence $T_3^1 T_3^2 T_2^1 T_1^1 T_4^1 T_1^2 T_3^3 T_3^4 T_2^2 T_4^2$ is a linear extension of the partial order described in Figure 1. (Let us notice that this linear extension does not respect real time order.)

Legal transaction The notion of legality is crucial for defining a consistency condition. It expresses the fact that a transaction does not read an overwritten value. More formally, given a linear extension \widehat{S} , a transaction T is *legal* in \widehat{S} if, for each $r_T(X)v \in T$, there is a committed transaction T' such that:

- $T' \rightarrow_S T$ and $w_{T'}(X)v \in T'$, and
- There is no transaction T'' such that $T' \rightarrow_S T'' \rightarrow_S T$ and $w_{T''}(X) \in T''$.

If all the transactions are legal, the linear extension \widehat{S} is legal.

In the following, a legal linear extension of a partial order, that models an execution of a set of transactions, is sometimes called a *sequential witness* (or witness) of that execution.

Causal past of a transaction Given a partial order \widehat{PO} defined on a set of transactions, the *causal past* of a transaction T , denoted $past(T)$, is the set including T and all the transactions T' such that $T' \rightarrow_{PO} T$. Let us observe that, if T is an aborted transaction, it is the only aborted transaction contained in $past(T)$.

3 A framework for defining a family of STM consistency conditions

The previous definitions constitute a general framework from which it is possible to state consistency conditions suited to STM systems. These conditions concern the safety property of a STM system: they allow to decide whether a given execution is correct or not⁶.

3.1 Considering only the committed transactions

This section uses the previous framework to state classical consistency conditions encountered in the context of database transactions. As they all impose constraints only on the transactions that commit, this section considers only the c-history $\widehat{CH} = (CH, \rightarrow_{CH})$ associated with a set of transactions.

Serializability [7, 27] A given transaction execution, modeled by the c-history $\widehat{CH} = (CH, \rightarrow_{CH})$, is *serializable* if it has a linear extension $\widehat{CS} = (CH, \rightarrow_{CS})$ that is legal.

The linear extension \widehat{CH} represents a sequential witness execution on which all the committed transactions agree. If each transaction is made up of a single read or a single write (in which case, no transaction should be aborted), serializability boils down to the consistency condition called *sequential consistency* [24].

Strict Serializability [7, 27] A given transaction execution, modeled by the c-history $\widehat{CH} = (CH, \rightarrow_{CH})$, is *strict serializable* if it has a linear extension $\widehat{CS} = (CH, \rightarrow_{CS})$ that (1) is legal, and (2) respects the real time order on the committed transactions (i.e., $T1 \rightarrow_{RT} T2 \Rightarrow T1 \rightarrow_{CS} T2$). (So, when compared to serializability, the *strictness* attribute adds consistency with respect to real time.)

If each transaction consists of a single operation on a predefined object defined by a sequential specification, e.g. a stack, a queue, or a register (then, the transactions are no longer dynamically defined), the strict serializability condition boils down to *linearizability* [18] (and, as before, no transaction should be aborted). (It is shown in [28] that sequential consistency can be interpreted as “lazy linearizability”).

Causal consistency [29] Causally consistent transactions have been introduced for collaborative applications where the fact that all the transactions have to agree on the very same linear extension is a stronger consistency requirement than necessary.

A given transaction execution, modeled by the c-history $\widehat{CH} = (CH, \rightarrow_{CH})$, is *causally consistent* if, for every process p_i , there is a linear extension $\widehat{CS}_i = (CH, \rightarrow_{CS_i})$ that is legal.

The fundamental difference with serializability is that causal consistency allows each process p_i to have its own consistent view of the execution (as witnessed by \widehat{CS}_i). If (assuming there are n processes) $\widehat{CS}_1 = \dots = \widehat{CS}_n$ we obtain serializability. On another side, if each transaction is reduced to a single read or a single write, (1) as indicated before no transaction needs then to be aborted, and (2) we obtain a causally consistent read/write shared memory [2]. (A consistency condition, called *causal serializability*, that lies between serializability and causal consistency is described in [29]. This condition is causal consistency plus the property that, for every object X , all the transactions see the write operations on X in the same order.)

⁶These conditions do not state when a transaction has to commit or abort. For the interested reader, a property, named *obligation* that forces a transaction to commit at least in “good circumstances” is presented in [21], together with a STM protocol that ensures it. An ideal STM system should never abort a transaction unless necessary for correctness. A corresponding metric, called *permissiveness*, has been introduced in [13] to compare STM systems (intuitively a STM system A is more permissive than a STM system B if, among the transactions that could be committed in a given execution, A aborts less transactions than B).

3.2 Considering all the transactions

The previous consistency conditions (used mainly for database transactions) place no requirements on the aborted transactions. It appears that, in the context of STM, these conditions can be too weak. A stronger consistency condition has recently been informally introduced in [11]. It states that the values read by any transaction, whatever its fate (commit or abort), must be mutually consistent. This condition has been given the name *opacity* in [14] where it is formally defined and deeply investigated⁷.

So, given an execution of a set of transactions, and its complete history $\widehat{CAH} = (CAH, \rightarrow_{CAH})$, this section uses the proposed framework not only to redefine opacity, but also to present new conditions weaker than opacity but strong enough to be meaningful and practically relevant. Basically, a condition has to formulate precisely a consistency notion for the set of values read by a (committed or aborted) transaction.

Opacity [14] A given execution ca-history $\widehat{CAH} = (CAH, \rightarrow_{CAH})$ is *opaque* if it has a linear extension $\widehat{CAS} = (CAH, \rightarrow_{CAS})$ that (1) is legal and (2) respects the real time order on all the transactions.

It is easy to see that opacity is strict serializability applied to all the transactions (an aborted transaction being reduced to a read prefix, as indicated in previous sections). So, in the following we call it *real time opacity*. STM protocols ensuring such an opacity consistency can be found in [11, 19, 30].

Virtual (or logical) time opacity The idea of this new consistency condition is to weaken opacity by not demanding its witness linear extension to comply with real time.

To be realistic (and meaningful) we have to ask the witness linear extension \widehat{CAS} to respect the order in which every process has issued its transactions (committed and aborted). To that end let us define for each process p_i the relation \rightarrow_i as follows: $T1 \rightarrow_i T2$ if both have been issued by p_i and $T1$ has been issued first. We are now in order to define virtual (or logical) time opacity.

A given execution ca-history $\widehat{CAH} = (CAH, \rightarrow_{CAH})$ is *virtual time opaque* if it has a linear extension $\widehat{CAS} = (CAH, \rightarrow_{CAS})$ that (1) is legal and (2) respects the relation “ \rightarrow_i ” of each process p_i . (It is easy to see that virtual time opacity is serializability applied to both the committed transactions and the appropriate read prefixes of the aborted transactions.)

A family of new consistency conditions: virtual world consistency Real time or virtual time opacity requires that all the transactions (be them committed or aborted) see the same witness execution \widehat{CAS} that complies with the (real or virtual) time notion considered. Weaker and meaningful consistency definitions that take into account aborted transactions are actually possible, and even desirable for STM systems. More precisely, we obtain the following family of consistency conditions.

- For the committed transactions: Either serializability or strict serializability can be considered.
- An aborted transaction T is *virtual world consistent* if there is a linear extension \widehat{S}_T of the partial order $past(T)$ that is legal.

An execution of a set of transactions is *virtual world* (resp., *strong virtual world*) consistent if (1) all the committed transactions are serializable (resp., strict serializable), and (2) each aborted transaction is *virtual world* consistent.

Let us observe that the witness \widehat{S}_T (from which T has been suppressed) is not required to be a prefix of the legal linear extension associated with the whole set of the committed transactions. It is easy to see that, while virtual world consistency is weaker than opacity, it remains a meaningful consistency condition as it requires that the object values read by each aborted transaction be mutually consistent.

⁷Let us observe that opacity rules out implementations that would allow domino effect and cascading aborts.

The idea that underlies this family of consistency conditions is the following. It guarantees that, in addition to the committed transactions, every aborted transaction reads values from a consistent global state of the shared memory. This state is consistent in the sense that, for each aborted transaction T , it appears in some legal history that is a witness for T . This does not mean that this state has really appeared in the shared memory; it only means that, from the point of view of the aborted transaction, the execution could have passed through this state. Hence, the name *virtual world* consistency. The important point is here that each of several aborted transactions T_1 (T_2 , etc.), sees a consistent global state (from which it reads the values of the objects in its read set) as given by a linear extension \widehat{S}_{T_1} (\widehat{S}_{T_2} , etc.): each witness linear extension represents a possible “virtual world” that can be different from the other witness linear extensions.

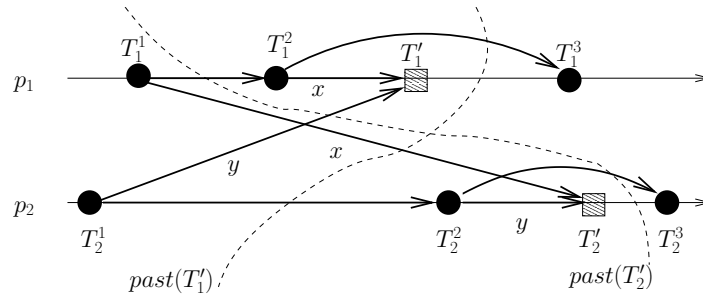


Figure 3: Examples of causal pasts

A simple example appears in Figure 3. The process p_1 has issued four transactions, the three of them (T_1^1 , T_1^2 and T_1^3) that have been committed have written x , while the one that has been aborted has read x and y . Similarly, the three transactions of p_2 that have been committed have written y and the one that has been aborted has read x and y . The causal pasts of T_1' and T_2' are indicated on the figure (transactions on the left part of the corresponding dotted line). It is easy to see that $S_{T_1'} = T_1^1 T_2^1 T_1^2 T_1^3$ is a legal linear extension for $past(T_1')$, while $S_{T_2'} = T_2^1 T_1^1 T_2^2 T_2^3$ is a legal linear extension for $past(T_2')$. Both T_1' and T_2' have read their values from consistent global states, but while each is consistent, these global states cannot occur in the same execution. They belong to two different -but consistent- virtual worlds (in the example, one of these virtual worlds actually occurs, the one associated with the linear extension $T_2^1 T_1^1 T_1^2 T_2^3$). (It is easy to see that the linear extension $\widehat{S}_{T_1'} = T_2^1 T_1^1 T_1^2 T_1^3$ satisfies the real time requirement while $\widehat{S}_{T_2'} = T_2^1 T_1^1 T_2^2 T_2^3$ does not as it misses T_1^2 that precedes T_2^2 in real time order.)

One of the main interests of virtual world consistency lies in the fact that it prevents bad phenomena from occurring without requiring all the transactions (committed or aborted) to agree on the same witness execution. Let us assume that, when executed alone and it reads a consistent state of the objects, each transaction behaves correctly (e.g. it does not entail a division by 0, does not enter an infinite loop, etc.). As, due to the virtual world consistency condition, no transaction (committed or aborted) reads from an inconsistent state, it cannot behave incorrectly despite concurrency; it can only be aborted. This is a first class requirement for transactional memories.

3.3 Transaction systems vs message-passing systems

If we consider each transaction as an atomic event produced by a process, and the read-from relation as corresponding to message exchanges (carrying new values of the objects) the partial order on a set of transactions $\widehat{PO} = (PO, \rightarrow_{PO})$ is the equivalent of Lamport’s *happened before* relation [23]. Views of distributed computations where base events are “packed together” to define “bigger atomic events” (similarly to -but

differently from- transactions) are presented in [15] and [22]. Similarly, the notion of causal past associated with a transaction is nothing else than the causal past notion associated with events in message-passing distributed computations [26, 31].

In the same vein, the existence of a legal extension does correspond to the notion of *consistent observation* as defined in [31]. Finally, the consideration of several linear extensions of a partial order corresponds to the existence of several consistent sequential observations of the same message-passing execution. Let us recall that, in a message-passing distributed system, if several processes simultaneously launch independent global state computations, these processes can obtain different consistent global states [9] (this has given rise to the *possibly* and *definitely* modalities for detecting unstable properties on the global states of a distributed application [6, 10]). Here, the “equivalent” is the possible existence of several different but consistent virtual worlds.

4 A STM protocol that ensures virtual world consistency

This section presents an algorithm that consumes on-line the text of transactions submitted by processes and produces runs that satisfy virtual world consistency, which (as we have seen) is stronger than serializability (as it adds constraints on the aborted transactions) but weaker than both real time opacity and virtual time opacity (as it does not require that the very same linear extension be a witness for all the aborted transactions⁸).

4.1 The STM system interface

The STM system provides the transactions with four operations denoted $\text{begin}_T()$, $X.\text{read}_T()$, $X.\text{write}_T()$, and $\text{try_to_commit}_T()$, where T is a transaction issued by a process p_i , and X a base object.

- $\text{begin}_T()$ is the first operation issued by a process p_i when it starts a new transaction T .
- $X.\text{read}_T()$ is invoked by the transaction T to read the base object X . That operation returns a value of X or the control value *abort* (in which case T is aborted).
- $X.\text{write}_T(v)$ is invoked by the transaction T to update X to the new value v . As we will see, that operation never forces a transaction to immediately abort. It always returns *ok*.
- If a transaction attains its last statement (as defined by the user) it executes $\text{try_to_commit}_T()$. That operation decides the fate of T by returning *commit* or *abort*. (Let us notice, a transaction T that invokes $\text{try_to_commit}_T()$ has not been aborted during an invocation of $X.\text{read}_T()$.)

4.2 The STM system variables

Shared control variables To implement the previous operations, the STM system uses the following atomic control variables. The shared objects accessed by the transactions, and the shared control variables -i.e., all the variables kept in shared memory- are denoted with uppercase letters. (Local variables will be denoted with lowercase letters with an appropriate process/transaction index.)

- A logical clock denoted *CLOCK*. Initialized to 0, it can be read and increased.
- A lock per base object X . Locks are assumed to be fair (assuming each lock is eventually released, every transaction that requires a lock eventually gets it).
- A set RS_X per base object X . This set, initialized to \emptyset , contains the ids of the transactions that have read X since the last update of X . A transaction adds its id to RS_X to indicate a possible read/write **conflict**.

⁸The structure of the algorithm is similar to the one presented in [19] that satisfies opacity.

- Each base object X is made up of two fields: $X.value$ denotes its current value, while $X.date$ denotes the logical date at which that value has been written.
- A control variable MAX_DATE_T , initialized to $+\infty$, is associated with each transaction T . It keeps the smallest date at which an object read by T has been overwritten. Let us notice that there are n such variables, namely one per process p_i .
- A shared token SEQ , that contains a sequence of 4-uples denoted $\langle ser_date, rs, ws, commit_date \rangle$. This token is used by a transaction when it invokes $try_to_commit_T()$. To keep it consistent, the accesses to this token are protected by a lock. SEQ contains a list (which can be garbage collected) of 4-uples, one for each committed transaction T , the meaning of which is the following:
 - rs and ws are the read set and the write set of T , respectively.
 - ser_date and $commit_date$ are two dates defined from the clock: ser_date is the serialization date of T , while $commit_date$ is the date at which it has been committed. The serialization dates provide the witness linear extension for the committed transactions [18].
 It is important to see that, contrarily to protocols that ensure opacity (such as [11, 19]), the serialization date and the commit date of a committed transaction are not required to be the same. We can have $ser_date \leq commit_date$ (this allows more transactions to commit).

Local control variables Each process manages the following local variables. To make the presentation clearer, one is indexed with the process name, while the others are indexed with the transaction name. T denotes the transaction (if any) currently issued by p_i and not yet terminated.

- $last_commit_date_i$ is the commit date of the last committed transaction issued by p_i .
- lrs_T and lws_T are sets where p_i keeps track of the objects read and written by T , respectively.
- For each object accessed by T , p_i keeps a copy $lc(X)$ in its local memory. This copy has two fields denoted $lc(X).value$ and $lc(X).date$.
- min_date_T contains the greatest date of the objects read so far by T . It is initialized to the current value of $last_commit_date_i$. When T is about to read a new object from the shared memory, the pair (min_date_T, MAX_DATE_T) allows p_i to check if adding this new read to T 's current snapshot preserves or not consistency.
- The pair of local variables $(to_commit_T, to_write_T)$ is used by p_i when it executes $try_to_commit_T()$. Their meaning is as follows: to_commit_T is a boolean that, when evaluated to true, indicates that (1) the transaction T can be committed, and (2) only the objects in the subset $to_write_T \subseteq lws_T$ have to be written in the shared memory.

4.3 The STM protocol

The algorithms implementing the four operations that constitute the STM system ($begin_T()$, $X.read_T()$, $X.write_T(v)$ and $try_to_commit_T()$) are described in Figure 4. They rely on two basic ideas:

- As in other protocols (e.g., STM or discrete event simulation), one is to associate a time window with each transaction. If this window becomes empty, the transaction has to be aborted.
- The second one (as already announced) consists in not directing a transaction that commits to be serialized to its commit time. Distinguishing serialization time and commit time allows more transactions to be committed and (as we will see) can save write into the shared memory.

```

operation beginT():
(01) min_dateT ← last_commit_datei; MAX_DATET ← +∞.
=====
operation X.readT():
(02) if (there is no local copy of X) then
(03)   allocate local space lc(X) for a copy;
(04)   lock X; lc(X) ← X; RSX ← RSX ∪ {T}; unlock X;
(05)   lrsT ← lrsT ∪ {X};
(06)   min_dateT ← max(min_dateT, lc(X).date);
(07)   if (min_dateT > MAX_DATET) then return(abort) end if
(08) end if;
(09) return (lc(X).value).
=====
operation X.writeT(v):
(10) if (there is no local copy of X) then allocate local space lc(X) for a copy end if;
(11) lc(X).value ← v;
(12) lwsT ← lwsT ∪ {X};
(13) return (ok).
=====
operation try_to_commitT():
(14) lock all the objects in lrsT ∪ lwsT; lock the lock protecting SEQ and CLOCK;
(15) to_commitT ← true; to_writeT ← lwsT;
(16) for each < ser_date, rs, ws, c_date > ∈ SEQ such that min_dateT < c_date do
(17)   if ser_date ≤ min_dateT
(18)     then to_commitT ← to_commitT ∧ (ws ∩ lrsT = ∅)
(19)     else to_commitT ← to_commitT ∧ (rs ∩ lwsT = ∅); to_writeT ← to_writeT \ ws end if;
(20) end for;
(21) if (¬to_commitT) then release all the locks; return(abort) end if;
(22) for each T' ∈ ∪X ∈ to_writeT RSX do MAX_DATET' ← min(MAX_DATET', CLOCK) end for;
(23) CLOCK ← CLOCK + 1; commit_dateT ← CLOCK;
(24) for each X ∈ to_writeT do X ← (lc(X).value, commit_dateT); RSX ← ∅ end for;
(25) SEQ ← SEQ · < min_dateT, lrsT, lwsT, commit_dateT >;
(26) last_commit_datei ← commit_dateT;
(27) release all the locks;
(28) return(commitT).

```

Figure 4: A STM algorithm that satisfies virtual world consistency

The operation $X.\text{begin}_T()$ When it starts a new transaction T , p_i sets the initial values of its time window: min_date_T is set to $\text{last_commit_date}_i$ and MAX_DATE_T is set to $+\infty$. Then, min_date_T can only increase, while MAX_DATE_T can only decrease.

The operation $X.\text{read}_T()$ When T invokes $X.\text{read}_T()$, it obtains the value of X currently kept in the local memory if there is one (lines 02 and 09). Otherwise, T first allocates space in its local memory for a copy of X (line 03), obtains the value of X from the shared memory and updates RS_X accordingly (line 04). The update of RS_X allows T to announce a read/write conflict that will occur with the transactions that will update X . This line is the only place where possible future read/write conflicts are announced in the STM algorithm.

Then, T updates its local control variables lrs_T (line 05) and min_date_T (line 06) in order to keep them consistent. Finally, T checks its time window (line 07) to know if its snapshot is consistent. If the time window is empty, the value it has just obtained from the memory can make its current snapshot inconsistent and consequently T aborts.

The operation $X.write_T()$ The text of the algorithm implementing $X.write_T()$ is very simple. If there is no local copy of X , a corresponding space is allocated in the local memory (line 10); let us remark that this does not entail a read of X from the shared memory. Then, T updates the local copy of X (line 11), and records X in lws_T (line 12). It is important to notice that an invocation of $X.write_T()$ is purely local: it involves no access to the shared memory, and cannot entail an immediate abort of the corresponding transaction.

The operation $try_to_commit_T()$ This operation works as follows. First T locks all the objects in its read and write sets (according to a predefined total order in order to prevent deadlocks), plus the lock that protects SEQ and $CLOCK$ (line 14). Let us notice that if T 's time window was not empty after T 's last read operation, it cannot become empty later. This is due to the facts that transactions are committed in mutual exclusion and that a committing transaction T' cannot change the variable MAX_DATE_T of T to a value lower than the current value of $CLOCK$ (line 22 of T' 's try_to_commit operation), this value being necessarily at least as big as the dates of the objects already read by T . Thus, T 's time window cannot become empty after its last read operation.

Then, the process p_i checks to see if T can be serialized at the date min_date_T with its write taking effect at the next clock value, i.e. $CLOCK + 1$ (see lines 23-24). But, as the serialization dates of the committed transactions are different from their commit dates, we have to be careful that linearizing T at min_date_T does not falsify the consistency of the transactions that have been already committed (lines 15-21). To that end, min_date_T is compared to the serialization date (ser_date) of each committed transaction. Let $\langle ser_date, rs, ws, c_date \rangle$ be the 4-uple associated with such a transaction $T' \in SEQ$ (line 16). If $c_date < min_date_T$, linearizing T' at min_date_T cannot create problems for these transactions: the write of T' occurred at c_date , the writes of T occur at $CLOCK + 1$ and T' is “naturally” serialized before T . If $c_date > min_date_T$, the situation is different, and depends on the respective values of ser_date and min_date_T (Figure 5).

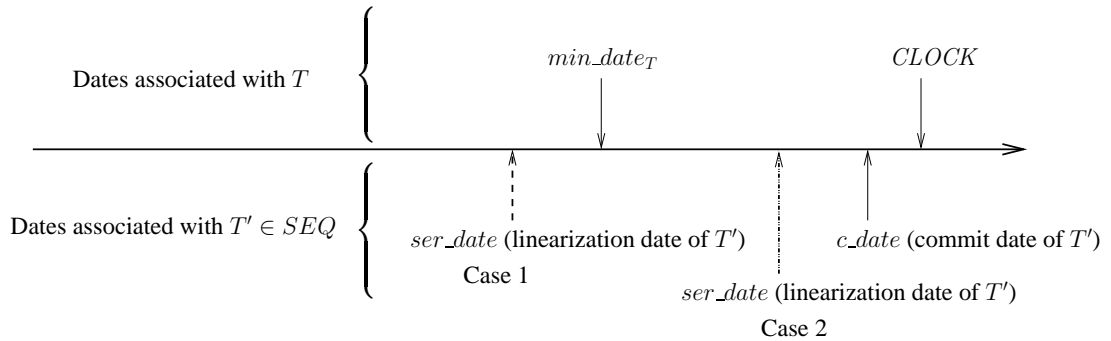


Figure 5: Respective position of min_date_T and ser_date

- Case 1: $ser_date \leq min_date_T$ (line 18). In that case, T can safely be serialized after T' if between min_date_T and the time at which T' has been committed (c_date), T' has not written into an object read by T , which is operationally checked by the predicate $ws \cap lrs_T = \emptyset$.
- Case 2: $ser_date > min_date_T$ (line 19). In that case, T can safely be serialized before T' if between min_date_T and the time ser_date at which T' is serialized, T has not overwritten the values read by T' , i.e., if $rs \cap lws_T = \emptyset$. Moreover, if this predicate is true, T does not have to write into the objects written by T' (as, in that case, T' has overwritten the objects in $lws_{T'} \cap lws_T$, and these objects have been written by T' at its commit date c_date). The local variable to_write is used to record the objects that T has to write.

If to_commit_T is false, T cannot be correctly serialized and is consequently aborted. Otherwise, it commits. The commit step is made up of the lines 23-26: T increases the clock, writes their new values into the objects $X \in to_write_T$ (and resets accordingly their RS_X sets), adds the 4-uple associated with T (namely $\langle min_date_T, lrs_T, lws_T, CLOCK \rangle$) to SEQ , and increases $last_commit_date_i$ to the present commit date (which is the current value of $CLOCK$). All the locks are finally released (line 27) and the value $commit$ is returned (line 28).

Garbage collecting SEQ A garbage collector can be implemented to eliminate the 4-uples of SEQ that become irrelevant. This can be done using a shared array (with one entry per process) that records the date min_date_T of each live transaction T . The garbage collector removes the transactions $\in SEQ$ whose commit date c_date is smaller than $\min(min_date_T \text{ such that } T \text{ is live})$. Note that the reads of the array do not have to be atomic (thereby, not demanding an expensive snapshot operation).

4.4 From virtual world to strong virtual world consistency

It appears that a simple modification to the algorithm described in Figure 4 provides a variant that guarantees strong virtual world consistency (which means that it ensures that the committed transactions are now *strict* serializable). This modification is the following: The local variable $last_commit_date_i$ of each process p_i is suppressed (so, line 26 disappears), and the update of min_date_T at line 01 becomes $min_date_T \leftarrow CLOCK$. (The word “serialization” has then to be replaced by “linearization” in the description of the protocol).

5 Proof of the protocol

The proof is in two parts. It is first shown that the set of committed transactions accepts a legal linear extension $\widehat{S} = (S, \rightarrow_S)$ that respects the partial order on the committed transactions (Section 5.2). It is then shown that each aborted transaction T reads from a global state that is consistent.

5.1 Preliminary definitions

- Considering a complete execution, \mathcal{C} denotes the set of all its committed transactions, and \mathcal{A} denotes the set of all its aborted transactions.

Let us observe that, given an execution of the STM system described in Figure 4, \mathcal{C} is exactly the set of transactions that define SEQ .

- Let us recall that the set $\widehat{H} = (H, <_H)$ (Section 2.2) is the total order on the set H of all the base events (begin, read, write and end) issued by the transactions.
- Let \rightarrow_{proc} be the partial order defined as follows: $T \rightarrow_{proc} T'$ if T and T' have been issued by the same process and T has been issued before T' (process order).
- Let \xrightarrow{X}_{rf} be the partial order defined as follows: $T \xrightarrow{X}_{rf} T'$ if T' reads from X the value v that has been written by T (read-from order), so we have $w_{T'}(X)v <_H r_T(X)v$. Moreover, $\rightarrow_{rf} = \bigcup_X \xrightarrow{X}_{rf}$.
- Let us recall that, when we consider the transaction partial order $\widehat{PO} = (PO, \rightarrow_{PO})$ defined in Section 2.3, we have $PO = \mathcal{C} \cup \mathcal{A}$ and \rightarrow_{PO} is the transitive closure of $(\rightarrow_{proc} \cup \rightarrow_{rf})$.

5.2 The committed transactions can be totally ordered

To obtain a legal linear extension $\widehat{S} = (S, \rightarrow_S)$, the total order \rightarrow_S is defined as follows. The transactions in SEQ are ordered by their *ser_date* dates. If two transactions have the same *ser_date* value, they are tie-broken by their *commit_date* values (which are unique, as *CLOCK* is increased at each successful *try_to_commit* operation, line 23).

Lemma 1 $\forall T \in SEQ : ser_date_T < commit_date_T$.

Proof

Let τ be the time just before the transaction T executes line 23 (where it increases *CLOCK* and defines its commit date). Let t be the value of *CLOCK* at time τ . We have the following.

- *CLOCK* can only increase, and can be increased by only one process at a time.
- At time τ , all the (logical) dates in the system associated with the date field of any object are $\leq t$. This follows from the previous item and lines 23-24 executed by a transaction when it updates an object.
- At time τ , the *min_date* $_{T'}$ local variable of any transaction T' (including T) is $\leq t$. This follows from the previous item and line 06 when executed by T . Hence, as *ser_date* $_T$ is the last value of the non-decreasing variable *min_date* $_T$, it is $\leq t$.

It follows from the last item that, after the transaction T has executed line 23, *commit_date* $_T = t + 1$, which proves the lemma. □_{Lemma 1}

Lemma 2 $\forall T, T' \in \mathcal{C} : (T \rightarrow_{proc} T') \Rightarrow (T \rightarrow_S T')$.

Proof Let p_i be the process executing T and T' . Let T'' be the last transaction committed by p_i when T' starts. We have the following.

The variable *min_date* $_{T'}$ is initialized at *commit_date* $_{T''}$ (lines 01 and 26). During the *try_to_commit*() operation, *ser_date* $_{T'}$ takes the value of *min_date* $_{T'}$ (line 25). Because *min_date* $_{T'}$ never decreases, we have *commit_date* $_{T''} \leq ser_date_{T'}$, and thus, because of Lemma 1, *commit_date* $_{T''} < commit_date_{T'}$. It follows from this fact and the definition of \rightarrow_S that $\forall T, T' \in \mathcal{C} : (T \rightarrow_{proc} T') \Rightarrow (T \rightarrow_S T')$, which proves the lemma. □_{Lemma 2}

Lemma 3 $\forall T, T' \in \mathcal{C} : (T \xrightarrow{X}_{rf} T') \Rightarrow (T \rightarrow_S T')$.

Proof As $T \xrightarrow{X}_{rf} T'$, and the write of a transaction occurs just before its commit time, we conclude that T commits before T' reads the value it has written in X . We consequently have the following.

$$\begin{aligned} T \rightarrow_{rf} T' &\Rightarrow commit_date_T \leq ser_date_{T'} \text{ (lines 24 by } T \text{ and 06 by } T'), \\ \text{(Due to Lemma 1)} &\Rightarrow ser_date_T < ser_date_{T'}, \\ \text{(By definition of the serialization order)} &\Rightarrow T \rightarrow_S T'. \end{aligned}$$

□_{Lemma 3}

Lemma 4 $\forall T, T' \in \mathcal{C} : (T \xrightarrow{X}_{rf} T') \Rightarrow (\nexists T'' : (w(X) \in T'') \wedge (T \rightarrow_S T'' \rightarrow_S T'))$.

Proof The proof is by contradiction. Let us suppose that such a transaction T'' exists. As $T \xrightarrow{X}_{rf} T'$ we also have

$$(w_T(X) <_H r_{T'}(X)) \wedge ((w_{T''}(X) <_H w_T(X)) \vee (r_{T'}(X) <_H w_{T''}(X))).$$

We consider two cases: $w_{T''}(X) <_H w_T(X)$ and $r_{T'}(X) <_H w_{T''}(X)$.

- Case $w_{T''}(X) <_H w_T(X)$.

It follows from this case assumption that T'' was committed before T , and we have:

$$(w_{T''}(X) <_H w_T(X) <_H r_{T'}(X) \wedge T \rightarrow_S T'') \Rightarrow (w_T(X) \notin T \text{ (line 19)}) \Rightarrow (T \not\xrightarrow{X}_{rf} T'),$$

which contradicts the initial assumption, and consequently proves the lemma.

- Case $r_{T'}(X) <_H w_{T''}(X)$.

If T' is committed before T'' , T'' is not allowed to commit (due to line 19). If T'' is committed before T' , T' is not allowed to commit either (due to line 18). It follows that such a transaction T'' cannot exist, which contradicts the initial assumption and concludes the proof of the lemma. $\square_{\text{Lemma 4}}$

Lemma 5 *The set \mathcal{C} of all committed transactions accepts a legal linear extension \rightarrow_S that respects the partial order \rightarrow_{PO} on these transactions.*

Proof The proof follows from the definition of the total order \rightarrow_S (serialization dates), and the Lemmas 2, 3 and 4. $\square_{\text{Lemma 5}}$

5.3 The aborted transactions read consistent values

We now prove that for each aborted transaction T , $past(T)$ has a legal linear extension that respects \rightarrow_{PO} . For a committed transaction T , let $commit_date_T$ be its $commit_date$ value (recorded in T 's entry in SEQ). Because $CLOCK$ always increases, we can use its value as a date. We use this “date view” of $CLOCK$ in the following way: date t corresponds to the time instant at which $CLOCK$ is increased and reaches value t . $commit_date_T$ is then the time at which T increases $CLOCK$ (line 23).

For a transaction T , let $MAX_T(t)$ be the value of MAX_DATE_T at $CLOCK$'s time t .

Lemma 6 $\forall T1, T2 \in \mathcal{C} : T1 \in past(T2) \Rightarrow commit_date_{T1} < commit_date_{T2}$.

Proof We consider three cases.

- Read-from order. Let us assume that $T1 \rightarrow_{rf} T2$. As $T2$ reads from $T1$, it follows that $T1$ has committed before $T2$. As $CLOCK$ is increased at each commit (line 23), we have $commit_date_{T1} < commit_date_{T2}$.
- Process order. Let us assume $T1 \rightarrow_{proc} T2$. As (1) $CLOCK$ is updated at each commit (line 23), and (2) $T1$ and $T2$ are from the same process, we have $commit_date_{T1} < commit_date_{T2}$.
- Transitivity. this case follows trivially from the previous ones.

It follows from these items that $\forall T1, T2 \in \mathcal{C} : T1 \in past(T2) \Rightarrow commit_date_{T1} < commit_date_{T2}$. $\square_{\text{Lemma 6}}$

Lemma 7 $\forall T \in \mathcal{C} \cup \mathcal{A} : \forall T' \in \mathcal{C} : (MAX_T(commit_date_{T'}) < commit_date_{T'}) \Rightarrow (T' \notin past(T))$.

Irisa

Proof It follows from $\forall T \in \mathcal{C} \cup \mathcal{A} : \forall T' \in \mathcal{C} : (MAX_T(commit_date_{T'}) < commit_date_{T'})$ that (1) $T' \not\rightarrow_{rf} T$ (due to line 07 executed by T and line 24 executed by T'), and (2) $T' \not\rightarrow_{proc} T$ (due to the fact that T is already started at time $commit_date_{T'}$, line 01 by T), from which we conclude that T' cannot be T 's immediate predecessor in the partial order \rightarrow_{PO} . The facts that MAX_DATE_T can only decrease (at line 22) and that $\forall T1, T2 \in \mathcal{C} : T1 \in past(T2) \Rightarrow commit_date_{T1} < commit_date_{T2}$ (Lemma 6) show that T' cannot be T 's predecessor in \rightarrow_{PO} , and thus $T' \notin past(T)$, which proves the lemma. $\square_{Lemma 7}$

Definition Let \rightarrow_T be the total order \rightarrow_S (defined on committed transactions) (1) restricted to the transactions in $past(T)$, and (2) augmented with the pairs (T', T) such that $\forall T' \in past(T) : T' \neq T \Rightarrow T' \rightarrow_T T$. Because T is the only aborted transaction in $past(T)$, \rightarrow_T is a total order on $past(T)$. Let PT be the set of the transactions in $past(T)$. $\hat{T} = (PT, \rightarrow_T)$ is then a linear extension of $past(T)$.

Lemma 8 $\forall T \in \mathcal{A}$, $past(T)$ accepts a legal linear extension that respects the partial order \rightarrow_{PO} .

Proof In order to prove that \hat{T} is legal and respects \rightarrow_{PO} , we have to prove that $\forall T1, T2 \in past(T)$:

- (1) $(T1 \rightarrow_{proc} T2) \Rightarrow (T1 \rightarrow_T T2)$,
- (2) $(T1 \rightarrow_{rf} T2) \Rightarrow (T1 \rightarrow_T T2)$, and
- (3) $\forall X : [(T1 \xrightarrow{X}_{rf} T2) \Rightarrow \nexists T3 \in past(T) : ((w(X) \in T3) \wedge (T1 \rightarrow_T T3 \rightarrow_T T2))]$.

We first show that $\forall T1, T2 \in past(T) : T1 \rightarrow_{proc} T2 \Rightarrow T1 \rightarrow_T T2$. If $T1$ and $T2$ are both committed transactions, Lemma 2 applies and the fact that \rightarrow_T respects \rightarrow_S shows that $T1 \rightarrow_{proc} T2 \Rightarrow T1 \rightarrow_T T2$. Because $\forall T' : T \rightarrow_{proc} T' \Rightarrow T' \not\rightarrow_{rf} T$ (T cannot read a value written by a transaction started after it ended), $past(T)$ does not contain any transaction T' such that $T \rightarrow_{proc} T'$. By definition of \rightarrow_T , we have $T1 \in past(T) \cap \mathcal{C} \Rightarrow T1 \rightarrow_T T$, thus $T1 \rightarrow_{proc} T2 \Rightarrow T1 \rightarrow_T T2$.

We show then that $\forall T1, T2 \in past(T) : T1 \rightarrow_{rf} T2 \Rightarrow T1 \rightarrow_T T2$. If $T1$ and $T2$ are both committed transactions, Lemma 3 applies and the fact that \rightarrow_T respects \rightarrow_S shows that $T1 \rightarrow_{rf} T2 \Rightarrow T1 \rightarrow_T T2$. Because T is the only aborted transaction in $past(T)$, $\nexists T' : T \rightarrow_{rf} T'$, so it remains to consider only the case $T2 = T$. By definition of \rightarrow_T , we have $T1 \in past(T) \cap \mathcal{C} \Rightarrow T1 \rightarrow_T T$, thus $T1 \rightarrow_{rf} T2 \Rightarrow T1 \rightarrow_T T2$.

We show now that $\forall T1, T2 \in past(T) : \forall X : T1 \xrightarrow{X}_{rf} T2 \Rightarrow \nexists T3 \in past(T) : (w(X) \in T3) \wedge (T1 \rightarrow_T T3 \rightarrow_T T2)$. If $T1, T2, T3 \in \mathcal{C}$, Lemma 4 shows that such a $T3$ cannot exist. Moreover, as T is an aborted transaction, $\nexists T' : T \rightarrow_{rf} T'$ and $\nexists X : w(X) \in T$. Hence, the only case to consider is $T2 = T$.

Due to the lock on *CLOCK* and *SEQ*, the committed transactions are committed sequentially. Consequently, there is a total order on committed transactions (denoted \rightarrow_C) defined by the values of their variables *commit_date* (as recorded in *SEQ*).

According to the order on $T1$ and $T3$ with respect to \rightarrow_C , we consider two cases, namely, $T1 \rightarrow_C T3$ and $T3 \rightarrow_C T1$. In both cases, due to the assumptions stated in the item (2) we are proving, we also have $T1 \xrightarrow{X}_{rf} T$ and $T1 \rightarrow_T T3$.

- Case $T1 \xrightarrow{X}_{rf} T$, $T1 \rightarrow_T T3$, and $T1 \rightarrow_C T3$ ($T1$ is committed before $T3$). We have

$$\begin{aligned}
T1 \xrightarrow{X}_{rf} T &\Rightarrow T \in RS_X \text{ at time } commit_date_{T3}, \\
&\text{(because } T3 \text{ executes the lines 22-23 after } T \text{ executes line 04)} \\
&\Rightarrow MAX_T(commit_date_{T3}) < commit_date_{T3}, \\
&\Rightarrow T3 \notin past(T) \text{ (Lemma 7), from which it follows that this case cannot occur.}
\end{aligned}$$

- Case $T1 \xrightarrow{X}_{rf} T$, $T1 \rightarrow_T T3$ and $T3 \rightarrow_C T1$ ($T3$ is committed before $T1$).

In that case, it follows from $T3 \rightarrow_C T1$ that, when $T1$ executes line 19, X is suppressed from to_write_{T1} , and consequently $w(X) \notin T1$. Hence, we cannot have $T1 \xrightarrow{X}_{rf} T$, which contradicts the assumption. As previously, this case cannot occur, which completes the proof of the lemma.

□*Lemma 8*

Theorem 1 *The algorithm described in Figure 4 ensures the virtual world consistency condition.*

Proof The proof follows from the Lemmas 5 and 8.

□*Theorem 1*

5.4 On the non-triviality of the protocol

Due to the predefined order used by the transaction to acquire locks, no transaction can deadlock, and consequently, every transaction terminates (i.e., commits or aborts). Moreover, it is possible to show that, in an execution with an infinite number of transactions, an infinite number of transactions are committed, which means that the proposed STM system is not trivial (a trivial STM system has executions where all the transactions are aborted).

6 Conclusion

This paper was motivated by the definition of consistency conditions for software transactional memories. Its first contribution is the presentation of a simple but general framework from which a family of consistency conditions can be defined. This family includes the classical consistency conditions encountered in traditional transactions systems (serializability and strict serializability) [7, 27], and the opacity condition that has been designed for STM applications [11, 14].

The second contribution is the definition (from this framework) of a new consistency condition, called *virtual world consistency*. This new condition demands that (1) the committed transactions be serializable (or strict serializable), and (2) each aborted transaction reads values from a global state it perceives as consistent. Hence, while both virtual world consistency and opacity require the aborted transactions to read consistent snapshot of values, virtual world consistency is weaker as it does not require that the committed and the aborted transactions agree on a unique sequence of consistent global states. Two transactions that abort can see two different (and possibly incompatible) consistent global states. This new condition is relevant for a large class of STM applications as no transaction sees an inconsistent global state (it is weaker than opacity while preserving its spirit).

The third contribution of the paper is a protocol that implements an instance of the virtual world consistency condition (the one where committed transactions have to be serializable). Interestingly enough, this protocol uncouples the serialization time and the commit time of every transaction (thereby providing additional flexibility that can allow more transactions to commit). From a theoretical point of view, it would be interesting to know if such an uncoupling is possible or impossible for the protocols that implement opacity.

References

- [1] Afek Y., Attiya H., Dolev D., Gafni E., Merritt M. and Shavit N., Atomic Snapshots of Shared Memory. *Journal of the ACM*, 40(4):873-890, 1993.

- [2] Ahamad M., Neiger G., Burns J.E., Kohli P. and Hutto Ph.W., Causal Memory: Definitions, Implementation, and Programming. *Distributed Computing*, 9(1):37-49, 1995.
- [3] Attiya H., Needed: Foundations for Transactional Memory. *ACM Sigact News, Distributed Computing Column*, 39(1):59-61, 2008.
- [4] Attiya H., Guerraoui R. and Ruppert E., Partial Snapshot Objects. *Proc. 20th ACM Symposium on Parallel Algorithms and Architectures (SPAA'08)*, ACP Press, ACM Press, pp. 336-343, 2008.
- [5] Aydonat U. and Abdelrahman T.S., Serializability of Transactions in Software Transactional Memory. *Proc. 3rd ACM Workshop on Transactional Computing (TRANSAC'08)*, ACP Press, ACM Press, pp. 91-100, 2008.
- [6] Babaoğlu Ö. and Marzullo K., Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms. Chapter 4 in "Distributed Systems". ACM Press, Frontier Series, pp 55-93, 1993.
- [7] Bernstein Ph.A., Shipman D.W. and Wong W.S., Formal Aspects of Serializability in Database Concurrency Control. *IEEE Transactions on Software Engineering*, SE-5(3):203-216, 1979.
- [8] Cachopo J. and Rito-Silva A., Versioned Boxes as the Basis for Transactional Memory. *Science of Computer Programming*, 63(2):172-175, 2006.
- [9] Chandy K.M. and Lamport L., Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Operating Systems*, 3(1):63-75, 1985.
- [10] Cooper R. and Marzullo K., Consistent Detection of Global Predicates. *Proc. ACM/ONR Workshop on Parallel and Distributed Debugging*, ACM Press, pages 167-174, 1991.
- [11] Dice D., Shalev O. and Shavit N., Transactional Locking II. *Proc. 20th Int'l Symposium on Distributed Computing (DISC'06)*, Springer-Verlag, LNCS #4167, pp. 194-208, 2006.
- [12] Felber P., Fetzer Ch., Guerraoui R. and Harris T., Transactions are coming Back, but Are They The Same? *ACM Sigact News, Distributed Computing Column*, 39(1):48-58, 2008.
- [13] Guerraoui R., Henzinger Th.A. and Singh V., Permissiveness in Transactional Memories. *Proc. 22th Int'l Symposium on Distributed Computing (DISC'08)*, Springer-Verlag, LNCS #5218, pp. 305-319, 2008.
- [14] Guerraoui R. and Kapałka M., On the Correctness of Transactional Memory. *Proc. 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'08)*, ACM Press, pp. 175-184, 2008.
- [15] Hélary J.-M., Mostéfaoui A. and Raynal M., Interval Consistency of Asynchronous Distributed Computations. *Journal of Computer and System Sciences*, 64(2):329-349, 2002.
- [16] Herlihy M.P. and Luchangco V., Distributed Computing and the Multicore Revolution. *ACM SIGACT News*, 39(1): 62-72, 2008.
- [17] Herlihy M.P. and Luchangco V., Moir M. and Scherer III W.N., Software Transactional Memory for Dynamic-Sized Data Structures. *Proc. 22th ACM Symposium on Distributed Computing (PODC'03)*, ACM Press, pp. 92-101, 2003.
- [18] Herlihy M.P. and Wing J.M., Linearizability: a Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463-492, 1990.
- [19] Imbs D. and Raynal M., A Lock-based STM Protocol that Satisfies Opacity and Progressiveness. *12th Int'l Conference On Principles Of Distributed Systems (OPODIS'08)*, Springer-Verlag LNCS #5401, pp. 226-245, 2008.
- [20] Imbs D. and Raynal M., Help When Needed, but No More: Efficient Read/Write Partial Snapshots. *Tech Report*, #1907, 26 pages, IRISA, Université de Rennes (France), 2008.

- [21] Imbs D. and Raynal M., Provable STM Properties: Leveraging Clock and Locks to Favor Commit and Early Abort. *Proc. 10th Int'l Conference on Distributed Computing and Networking (ICDCN'09)*, Springer-Verlag, LNCS #5408, pp. 67-78, January 3-6, 2009.
- [22] Kshemkalyani A., A Framework for Viewing Atomic Events in Distributed Computations. *Theoretical Computer Science*, 196:45-70, 1998.
- [23] Lamport L., Time, Clocks and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558-565, 1978.
- [24] Lamport L., How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C28(9):690-691, 1979.
- [25] Larus J. and Kozyrakis Ch., Transactional Memory: Is TM the Answer for Improving Parallel Programming? *Communications of the ACM*, 51(7):80-89, 2008.
- [26] Mattern F., Virtual Time and Global States in Distributed Computations. *Proc. Int'l Workshop on Parallel and Distributed Algorithms*, Nort-Holland, (Cosnard, Quinton, Raynal and Robert, Eds), pp. 215-226, 1989.
- [27] Papadimitriou Ch.H., The Serializability of Concurrent Updates. *Journal of the ACM*, 26(4):631-653, 1979.
- [28] Raynal M., Sequential Consistency as Lazy Linearizability. *BA. Proc. 14th ACM Symposium on Parallel Algorithms and Architectures (SPAA'02)*, ACM Press, pp. 151-152, 2002.
- [29] Raynal M., Thia-kime G. and Ahamad M., From serializable to causal transactions. *BA. Proc. 15th ACM Symposium on Distributed Computing (PODC'96)*, ACM Press, pp. 310, 1996. Full version: From serializable to causal transactions for collaborative applications. *Proc. 23th EUROMICRO Conference*, IEEE Computer Press, pp. 314-321, 1997.
- [30] Riegel T., Fetzer C. and Felber P., Time-based Transactional Memory with Scalable Time Bases. *Proc. 19th annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'07)*, ACM Press, pp. 221-228, 2007.
- [31] Schwarz R. and Mattern F., Detecting Causal Relationship in Distributed Computations: in Search of the Holy Grail. *Distributed Computing*, 7:149-174, 1993.
- [32] Scott L.M., Sequential Specification of Transactional Memory Semantics. *Proc. First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT'06)*, ACM Press, 2006.
- [33] Shavit N. and Touitou D., Software Transactional Memory. *Distributed Computing*, 10(2):99-116, 1997.
- [34] Spear M.F., Marathe V.J., Scherer III W.N. and Scott M.L., Conflict Detection and Validation Strategies for Software Transactional Memory. *Proc. 20th Symposium on Distributed Computing (DISC'06)*, Springer-Verlag #4167, pp. 179-193, 2006.

A A variant

Aim This section presents a variant of the algorithm described in Figure 4 where a transaction T that commits is not required to be serialized at its min_date_T date. This allows transactions that would have been aborted with the base algorithm to commit. This is obtained at the price of a more involved algorithm.

The modification The modified algorithm is presented in Figure 6. The operations $begin_T()$, $X.read_T()$ and $X.write_T(v)$ are identical (they are kept on the figure for completeness). Only the algorithm associated with the operation $try_to_commit_T()$ has to be modified. Its code relies heavily on the value of $MAX_DATE_T = +\infty$. More precisely, we have the following.

- $MAX_DATE_T = +\infty$.
In that case, the transaction T is committed without any other test ($MAX_DATE_T = +\infty$ means that no value read by T has been overwritten). T is then serialized using the current value of $CLOCK$ as ser_date_T (lines 16-19, similarly to lines 22-25 in the base algorithm; as shown by line 19, we then have $ser_date_T = commit_date_T - 1$).
- $MAX_DATE_T \neq +\infty$.
In that case, min_date_T and MAX_DATE_T are updated according to the transactions in SEQ . As far as the serialization time if T is concerned, we have the following.
 - T has to be serialized before any transaction T' such that T has read the value of an object before T' overwrote it. MAX_DATE_T is then updated accordingly (line 22).
 - T has to be serialized after any transaction T' such that T writes an object whose value has been read or written by T' ; min_date_T is then updated accordingly (line 23).

Then, if T 's time window becomes empty, T is aborted (line 25). Otherwise, T is committed (lines 26-29, again similarly to lines 22-25 in the base algorithm).

```

operation beginT():
(01) min_dateT ← last_commit_datei; MAX_DATET ← +∞.
=====
operation X.readT():
(02) if (there is no local copy of X) then
(03)   allocate local space lc(X) for a copy;
(04)   lock X; lc(X) ← X; RSX ← RSX ∪ {T}; unlock X;
(05)   lrsT ← lrsT ∪ {X};
(06)   min_dateT ← max(min_dateT, lc(X).date);
(07)   if (min_dateT > MAX_DATET) then return(abort) end if
(08) end if;
(09) return (lc(X).value).
=====
operation X.writeT(v):
(10) if (there is no local copy of X) then allocate local space lc(X) for a copy end if;
(11) lc(X).value ← v;
(12) lwsT ← lwsT ∪ {X};
(13) return (ok).
=====
operation try_to_commitT():
(14) lock all the objects in lrsT ∪ lwsT; lock the lock protecting SEQ and CLOCK;
(15) if MAX_DATET = +∞ then
(16)   for each T' ∈ ∪X ∈ lwsT RSX do MAX_DATET ← min(MAX_DATET', CLOCK) end for;
(17)   CLOCK ← CLOCK + 1; commit_dateT ← CLOCK;
(18)   for each X ∈ lwsT do X ← (lc(X).value, commit_dateT); RSX ← ∅ end for;
(19)   SEQ ← SEQ · < commit_dateT - 1, lrsT, lwsT, commit_dateT >
(20) else
(21)   for each < ser_date, rs, ws, c_date > ∈ SEQ such that min_dateT < c_date do
(22)     if (ws ∩ lrsT ≠ ∅) then MAX_DATET ← min(MAX_DATET, ser_date - 1) end if;
(23)     if ((rs ∪ ws) ∩ lwsT ≠ ∅) then min_dateT ← max(min_dateT, ser_date) end if
(24)   end for;
(25)   if (min_dateT > MAX_DATET) then release all the locks; return(abort) end if;
(26)   for each T' ∈ ∪X ∈ lwsT RSX do MAX_DATET ← min(MAX_DATET', CLOCK) end for;
(27)   CLOCK ← CLOCK + 1; commit_dateT ← CLOCK;
(28)   for each X ∈ lwsT do X ← (lc(X).value, commit_dateT); RSX ← ∅ end for;
(29)   SEQ ← SEQ · < min_dateT, lrsT, lwsT, commit_dateT >
(30) end if;
(31) last_commit_datei ← commit_dateT;
(32) release all the locks;
(33) return(commitT).

```

Figure 6: A variant that commits more transactions