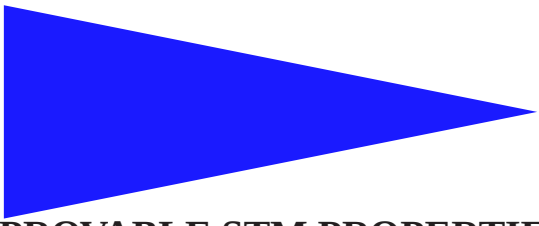
Provable STM Properties: Leveraging Clock and Locks to Favor Commit and Early Abort

Damien Imbs, Michel Raynal

I R I S A
INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES

PUBLICATION
INTERNE
N° 1894



PROVABLE STM PROPERTIES: LEVERAGING CLOCK AND LOCKS TO FAVOR COMMIT AND EARLY ABORT

DAMIEN IMBS MICHEL RAYNAL

ISSN 1166-8687

IRISA
CAMPUS UNIVERSITAIRE DE BEAULIEU - 35042 RENNES CEDEX - FRANCE

INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES
Campus de Beaulieu – 35042 Rennes Cedex – France
Tél. : (33) 02 99 84 71 00 – Fax : (33) 02 99 84 71 71
http://www.irisa.fr

Provable STM Properties: Leveraging Clock and Locks to Favor Commit and Early Abort

Damien Imbs* Michel Raynal**

Systèmes communicants
Projet ASAP

Publication interne n ° 1894 — Mai 2008 — 18 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 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.

The major part of papers devoted to STM systems address mainly their efficiency. Differently, this paper focuses on an orthogonal issue, namely, the design and the statement of a safety property. The only safety property that is usually considered is a global property involving all the transactions (e.g., conflict-serializability or opacity) that expresses the correction of the whole execution. Roughly speaking, these consistency properties do not prevent a STM system from aborting all the transactions. The proposed safety property, called *obligation*, is on each transaction taken individually. It specifies minimal circumstances in which a STM system must commit a transaction T. The paper proposes and investigates such an obligation property. Then, it presents a STM algorithm that implements it. This algorithm, which is based on a logical clock and associates a lock with each shared object, is formally proved correct.

Key-words: Atomic operation, Commit/abort, Concurrency control, Consistent global state, Lock, Opacity, Shared object, Software transactional memory, Transaction.

(Résumé : tsvp)

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

Centre National de la Recherche Scientifique
(UMR 6074) Université de Rennes 1 – Insa de Rennes

Institut National de Recherche en Informatique
et en Automatique – unité de recherche de Rennes

Favoriser la validation dans les mémoires transactionnelles logicielles

Résumé : Ce rapport présente une propriété de sûreté pour les mémoires transactionnelles logicielles. Le but de cette propriété est d'obliger le système à valider les transactions lorsque celles-ci apparaissent dans un certain contexte. Un protocole qui met en œuvre cette spécification est décrit et prouvé 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) has been proposed in [12]. It originates from the observation that the programmers were missing something when their applications are made up of concurrent processes that access sets of shared data structures (base objects). Roughly speaking, the main tools proposed to solve their synchronization problems were the locks and the notion of object multi-versioning (and associated version numbers): a lock allows preventing conflicting accesses to an object, while multi-versioning allows providing a process (without delaying it) with the appropriate version of an object (i.e., a version consistent with the other objects last values it has already obtained). Basically, locks are used to guarantee consistency (at the inherent price of possibly entailing delays), while versioning (when used) is employed to improve efficiency (by allowing to circumvent locking). The main problem with locks is that they are difficult to manage: locks controlling large sets of data reduce drastically parallelism, while locks controlling fine grain data are difficult to master and error-prone. Moreover, versioning can be very memory demanding.

The STM approach is a middleware approach that provides the programmers with the *transaction* concept. (As we will see, this concept is close but different from the notion of transactions encountered in databases [3].) More precisely, a process is designed as (or decomposed into) a sequence of transactions, each transaction being a piece of code that, while accessing any number of base objects, always appears as being executed atomically. 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, 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.

Of course, a solution in which a single transaction executes at a time trivially implements transaction atomicity but is 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 can direct it to abort some transactions in order to ensure 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).

Related work: STM consistency In the past years, several STM concepts have been proposed and several STM systems have been designed. They differ mainly in the consistency criterion (global safety property) they implement, and in the operational mechanisms their design is based on.

Two main consistency criteria have been considered so far, namely, serializability (as in databases), and opacity. Serializability [9] requires that the committed transactions appear as if they have been executed sequentially. This total order is not required to respect their commit order, nor even their real-time order. The two important points here are that serializability (1) places no requirement on the transactions that abort, and (2) is weaker than linearizability [6] (basically, linearizability requires that the total order respects the real-time order).

Differently, opacity places requirements on all the transactions (whatever their commit/abort fate), and involves linearizability. Suggested informally in [2], and given a name and formalized in [5], opacity is the addition of two properties. First, it requires that any transaction, whether it commits or aborts, always sees a mutually consistent state of the objects it accesses. (This means that a transaction has to be aborted before obtaining values that are not mutually consistent, or writing a value not consistent with the values it has read.) This means that an aborted transaction could be replaced by a maximal prefix (without write

operations and subsequent read operations on the same objects) that would commit. The second property lies in the fact that the committed transactions and the appropriate prefixes of the aborted transactions are linearizable: they can be totally ordered in a consistent way, and this order respects the real-time order on transactions.

Related work: Operational point of view Locks, versioning and (logical or physical) clocks are the main operational tools from which STM are built. We present here only a few STM systems that have been recently proposed. We focus on them because they all ensure the opacity criterion ([2] and [11] have been proposed before opacity has been formalized).

TL2 [2] is a single version, clock-based STM system. The logical clock is used to associate a date with each object, and ensures that the values read by a transaction define a consistent snapshot (belong to a consistent global state). TL2 satisfies the opacity property, but (in some cases) aborts a transaction T despite the fact that T is not conflicting with alive transactions. TL2C [1] is an extension of TL2 where the logical clock is distributed.

LSA-RT [11] is a STM system based on a real-time clock that manages several versions of each object. It does not use locks and satisfies the opacity property. As they are based on increasing clocks, both TL2 and LSA-RT have unbounded variables.

Differently from TL2 and LSA-RT, the protocol described in [7] does not use clocks and has only bounded variables. It requires a single version per object (as TL2), never aborts a write-only transaction, and aborts a transaction only in presence of conflicts (as LSA-RT, but differently from TL2). A comparison (from a property point of view) of this STM system with TL2 and LSA-RT is presented in [7].

Content of the paper: Favoring commit and providing early abort The design of nearly all the STM protocols proposed so far has mainly been driven by efficiency, measured as the number of transactions that commit per time unit, without taking into account the number of aborted transactions. But an aborted transaction can access the shared memory, consumes resources, and has to be restarted (usually as a new transaction). Very recently, a new efficiency measure has been proposed, that considers the ratio made up of the number of committed transactions divided by the total number of transactions [4]. On another side, nearly none of the protocols proposed so far has been formally proved correct. They are only explained, with a the sketch of an informal proof in the best cases.

The paper has several contributions. The first is a first step in proposing a provable commit property. While both an aborted transaction and a committed transaction terminate, an abort has to be considered as an unsuccessful termination while a commit is a successful termination. Considering this commit/abort dilemma, the paper introduces a transaction property that, when satisfied by a transaction T, requires that T commits. This property is designed incrementally. A property, called $P1(T)$, is first formulated that states whether the snapshot of object values read by the transaction T is consistent (i.e., could have been obtained by an atomic read of the shared memory). Then, this property is enriched to take into account the write operations issued by a transaction. This enriched property, called $P2(T)$, is such that $P2(T) \Rightarrow P1(T)$. It states whether both the snapshot of the values read by a transaction *and* its shared memory write operations could have been issued in a single atomic "macro-operation". These properties $P1(T)$ and $P2(T)$ are abstract in the sense that they are expressed in the model capturing the transaction executions. It is important to see that $P1(T)$ and $P2(T)$ are safety properties, that can be used to force a STM system to commit

transactions at least in "good circumstances". They are consequently called *obligation* properties[1]. An interesting side effect of $P2(T)$ is the fact it can be used to direct all the write-only transactions to commit.

Then, the paper presents its second contribution: a simple algorithm that implements a STM system satisfying the previous obligation properties. From an operational point of view, this algorithm is based on a logical clock (the logical clock could be replaced by a real-time clock or distributed real-time clocks as proposed and done in [11]; for simplicity, we only consider here a simple logical clock). It uses the following shared control variables: (1) a lock, a date and a read set are associated with each object, and (2) a date is associated with each transaction. Combined with the local control variables managed by each transaction, the shared control variables allow to express predicates that are correct implementations of the abstract properties $P1(T)$ and $P2(T)$ previously introduced. From an underlying design principle, a read of an object X from a transaction T announces only that X is read by T. Differently, when an update transaction T commits (and only at that time), T manages the read/write conflict it gives rise to, and announces possible future write/read conflicts. Moreover but not least, the algorithm is formally proved correct.

Finally, let us observe that the abort of a transaction is a stable property. It follows that, when the irrevocable decision to abort a transaction has been taken, there is no reason for that transaction to continue its execution: it has to be stopped as soon as possible. The proposed algorithm implements this observation in a simple way (at the additional price of possibly more shared memory accesses).

Roadmap The paper is made up of 6 sections. Section 2 presents the computation model and the obligation properties $P1()$ and $P2()$. Then, Section 3 presents a specification of a STM system, that takes into account the proposed obligation property for each transaction taken individually, and the opacity property (formalized in [5]) as the global consistency property linking all the transactions. Then, Section 4 presents the STM algorithm. Section 5 formally proves that it implements the previous specification. Finally, Section 6 concludes the paper.

The paper leaves open the problem of finding less constraining obligation properties (i.e., properties forcing more transactions to commit) and algorithms implementing them, the challenge being to find properties that do not require the implementation protocols to add "too many" control variables and not to be too synchronized (as these would not constitute acceptable solutions for a STM system).

2 Computation model and property statement

2.1 Computation model

Transaction As indicated, a transaction is a piece of code defined by the programmer. When (s)he defines a transaction T, the programmer considers that T is executed atomically (he does not have to worry about the management of the base objects accessed by the transaction). A transaction returns either commit or abort. 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. 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.

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

[1]This is similar to the specification of the *Non-Blocking Atomic Commit* problem. That problem has an *obligation* property capturing the circumstances in which a transaction must commit, namely when there is no failure and all the transactions have voted "yes".

- 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 is irrelevant $r_T(X)v$ is abbreviated $r_T(X)$.
- 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.

Given an execution, let H be the set of all the (begin, end, read and write) events generated by the transactions. As the events correspond to atomic operations, they can be totally ordered. It follows that, at the shared memory level, an execution can be represented by the pair $\widehat{H} = (H, <_H)$ where $<_H$ denotes the total ordering on its events. $\widehat{H}$ is called a *shared memory history*. As $<_H$ is a total order, it is possible to associate a unique "date" with each event in H. (In the following an event is sometimes used to denote its date.)

Types of conflict Two operations conflict if both access the same object and one of these operations is a write. Considering two transactions $T1$ and $T2$ that access the same object X, three types of conflict can occur. More specifically:

- Read/write conflict: $\mathit{conflict}(X, R_{T1}, W_{T2}) \stackrel{\text{def}}{=} \big(r_{T1}(X) <_H w_{T2}(X)\big)$.
- Write/read conflict: $\mathit{conflict}(X, W_{T1}, R_{T2}) \stackrel{\text{def}}{=} \big(w_{T1}(X) <_H r_{T2}(X)\big)$.
- Write/write conflict: $\mathit{conflict}(X, W_{T1}, W_{T2}) \stackrel{\text{def}}{=} \big(w_{T1}(X) <_H w_{T2}(X)\big)$.

History at the transaction level Let TR be the set of transactions issued during an execution. Let $\rightarrow_{TR}$ be the order relation defined on the transactions of TR as follows: $T1 \rightarrow_{TR} T2$ if $E_{T1} <_H B_{T2}$ ($T1$ has terminated before $T2$ starts). If $T1 \not\rightarrow_{TR} T2 \wedge T2 \not\rightarrow_{TR} T1$, we say that $T1$ and $T2$ are concurrent (their executions overlap in time). At the transaction level, that execution is defined by the partial order $\widehat{TR} = (TR, \rightarrow_{TR})$, that is called a *transaction level history* or a *transaction run*.

2.2 Two properties

This section investigates two properties that involve a transaction and the run in which it appears. These properties will be used in the specification of a STM system to force it to commit all the transactions that satisfy them. Given a run, let $\mathcal{C}$ denote the set of transactions that commit in that run.

2.2.1 A property ensuring snapshot consistency

Read a consistent snapshot Let a *snapshot* be a set of object values obtained by a transaction. A snapshot is *consistent* if there is a time t at which all the values it contains are the last values written in the shared memory before or at time t.

Let us consider the property, denoted $P1(T)$, defined as follows:

$$\forall T1, T2 \in \mathcal{C},\ \forall X1, X2 :\ \mathit{conflict}(X1, R_T, W_{T1}) \wedge \mathit{conflict}(X2, W_{T2}, R_T) :\ \ E_{T2} <_H B_{T1}.$$

Assuming a transaction reads an object at most once, the following theorem shows that, if $P1(T)$ is satisfied, the snapshot of values obtained by T is consistent.

Theorem 1 $P1(T) \Rightarrow$ *the snapshot obtained by* T *is consistent.*

Proof Let $\mathcal{C}_{WR}(T) = \{T_{WR}$ such that $\exists X : \mathit{conflict}(X, W_{T_{WR}}, R_T)\}$, i.e., $\mathcal{C}_{WR}(T)$ is the set of the transactions with which T has a write/read conflict. Similarly, let $\mathcal{C}_{RW}(T) = \{T_{RW}$ such that $\exists X : \mathit{conflict}(X, R_T, W_{T_{RW}})\}$, i.e., $\mathcal{C}_{RW}(T)$ is the set of the transactions that have a read/write conflict with T. Let us observe that $\mathcal{C}_{WR}(T) \cup \mathcal{C}_{RW}(T)$ is the set of the transactions that have written an object read by T.

Given a set S of events, let $\max_{<_H}(S)$ (resp., $\min_{<_H}(S)$) be the last (resp., first) event of S according to the total order $<_H$. Finally, let $\mathit{lower}(T) = \max_{<_H}\big(\{E_{T_{WR}} | T_{WR} \in \mathcal{C}_{WR}(T)\}\big)$, and let $\mathit{upper}(T) = \min_{<_H}\big(\{B_{T_{RW}} | T_{RW} \in \mathcal{C}_{RW}(T)\}\big)$.

It follows from the definition of write/read and read/write conflicts, that all the values read by T have been written into shared memory at time $\mathit{lower}(T)$ and have not been overwritten by time $\mathit{upper}(T)$. Moreover, due to the property $P1(T)$, we can conclude that $\mathit{lower}(T) <_H \mathit{upper}(T)$. It follows that the values read by T are the values most recently written into shared memory during the non-empty interval $[\mathit{lower}(T), \mathit{upper}(T)]$, which proves the theorem. $\square_{Theorem\ 1}$

As we will see in Section 5.2 (Corollary 2), a read-only transaction that satisfies the property $P1()$ can always be forced to commit.

2.2.2 A property ensuring atomicity

Atomicity A transaction T is *atomic* if (1) its reads (if any) define a consistent snapshot, and (2) its writes appear as if they have been executed immediately after the reads, "immediately" meaning "with no write operations (from other transactions) between its reads and writes". The transaction appears as if it has been executed at a given point of the time line, no two transactions being associated with the same point.

Let $P2(T)$ be the property defined as follows:

$$\forall T1, T2 \in \mathcal{C},\ \forall X1, X2 :\ \mathit{conflict}(X1, R_T, W_{T1}) \wedge \mathit{conflict}(X2, W_{T2}, R_T) :\ \ E_{T2} <_H E_T <_H B_{T1}.$$

Lemma 1 $\forall\, T : P2(T) \Rightarrow P1(T)$.

Proof Immediate from the definitions of $P1(T)$ and $P2(T)$ (suppress the event E_T in $P2(T)$). $\square_{Lemma\ 1}$

The following theorem shows that, if $P2(T)$ is satisfied, T is atomic.

Theorem 2 $P2(T) \Rightarrow T$ *is atomic.*

Proof Due to Lemma 1, the values read by T define a consistent snapshot. Moreover, it follows from $P2(T)$ that $\mathit{lower}(T) <_H E_T <_H \mathit{upper}(T)$. Therefore all the read operations of T appear as if they have been executed just before E_T and the write operations appear as if they have been executed at E_T, with no write operations from the other transactions in between, which proves the theorem. $\square_{Theorem\ 2}$

As we will see in Section 5.2 (Corollary 2), both the update transactions that satisfy the property $P2()$ and all the write-only transactions can always be forced to commit.

3 Problem specification

Safety properties specify which runs are correct. This paper considers two safety properties for a STM system. The first is opacity [5]. The second is an obligation property stating when a transaction is forced to commit. This section first presents opacity (in a way different from [5]), and then defines a STM specification. As already indicated, $\mathcal{C}$ is the set of transactions that commit. Let $\mathcal{A}$ denote the set of transactions that abort.

3.1 Preliminary definitions

A transaction history $\widehat{ST} = (ST, \rightarrow_{ST})$ is *sequential* if no two of its transactions are concurrent. Hence, in a sequential history, $T1 \not\rightarrow_{ST} T2 \Leftrightarrow T2 \rightarrow_{ST} T1$, thus $\rightarrow_{ST}$ is a total order. A sequential transaction history is *legal* if each of its read operations returns the value of the last write on the same object.

A sequential transaction history $\widehat{ST}$ is *equivalent* to a transaction history $\widehat{TR}$ if (1) $ST = TR$ (i.e., they are made of the same transactions (same values read and written) in $\widehat{ST}$ and in $\widehat{TR}$), and (2) the total order $\rightarrow_{ST}$ respects the partial order $\rightarrow_{TR}$ (i.e., $\rightarrow_{TR} \subseteq \rightarrow_{ST}$).

A transaction history $\widehat{AA}$ is *linearizable* if there exists a history $\widehat{SA}$ that is sequential, legal and equivalent to $\widehat{AA}$ [6].

3.2 The opacity property

Given a run $\widehat{TR} = \big(TR, \rightarrow_{TR}\big)$, and $T \in \mathcal{A}$, let $T' = \rho(T)$ be the transaction built from T as follows (ρ stands for "reduced"). As T has been aborted, there is a read or a write on a base object that entailed that abortion. Let $prefix(T)$ be the prefix of T that includes all the read and write operations on the base objects accessed by T until (but excluding) the read or write that entailed the abort of T. $T' = \rho(T)$ is obtained from $prefix(T)$ by replacing its write operations on base objects and all the subsequent read operations on these objects, by corresponding write and read operations on a copy in local memory. The idea here is that only an appropriate prefix of an aborted transaction is considered: its write operations on base objects (and the subsequent read operations) are made fictitious in $T' = \rho(T)$.

Finally, let $\mathcal{A}' = \{T' \mid T' = \rho(T) \wedge T \in \mathcal{A}\}$, and $\widehat{\rho(TR)} = \big(\rho(TR), \rightarrow_{\rho(TR)}\big)$ where $\rho(TR) = \mathcal{C} \cup \mathcal{A}'$ (i.e., $\rho(TR)$ contains all the transactions of $\widehat{TR}$ that commit, plus $\rho(T)$ for each transaction $T \in TR$ that aborts) and $\rightarrow_{\rho(TR)} = \rightarrow_{TR}$. Informally, opacity expresses the fact that a transaction never sees an inconsistent state of the base objects [2, 5]: the transactions in $\mathcal{C} \cup \mathcal{A}'$ can be consistently and totally ordered according to their real-time order. With the previous notation, opacity can be formally stated as follows:

- Opacity. $\widehat{\rho(TR)}$ is linearizable.

3.3 A STM specification

Similarly to serializability, opacity alone is too weak a safety property as it does not prevent trivial STM systems that would abort all transactions. (This observation was the main motivation in defining the properties $P1(T)$ and $P2(T)$.) Let *read_only(T)* be a predicate that is true iff T is a read-only transaction.

- Termination. Given a transaction T, let us assume that it terminates when executed in a concurrency-free context. Then, T terminates (commits or aborts) despite concurrency.
- Strong global consistency (Opacity). $\widehat{\rho(TR)}$ is linearizable.
- Obligation. $\forall\, T$: $\big(P1(T) \wedge read\_only(T)\big) \vee P2(T) \Rightarrow \big(T \in \mathcal{C}\big)$.

While the termination property is a liveness property (on a per transaction basis), global consistency and obligation are safety properties. The first is on the whole execution: it states that the execution is consistent. The second concerns each transaction taken individually: it states conditions where a transaction is obliged to commit. Those are characterized by the predicates $P1()$ and $P2()$ (consistency of the snapshot defined by the values read by a transaction, and atomicity of a transaction -its reads and writes can appear as having been executed without interfering operations from other transactions).

It is worth observing that a weaker specification of a STM system can be obtained by replacing the global consistency property by the following weaker property: $\widehat{\rho(TR)}$ is sequentially consistent [8, 10]

(both linearizability and sequential consistency require a "witness" equivalent legal sequential history $\widehat{ST}$, but only linearizability requires that $\widehat{ST}$ respects the real-time order defined by $\rightarrow_{TR}$, i.e., $\rightarrow_{TR} \subseteq \rightarrow_{ST}$).

4 A STM protocol based on clock and locks

This section presents an algorithm that implements an STM system. The next section proves that it satisfies the previous specification.

4.1 The STM system interface

The STM system provides the transactions with three operations denoted $X.\mathsf{read}_T()$, $X.\mathsf{write}_T()$, and $\mathsf{try\_to\_commit}_T()$, where T is a transaction, and X a base object.

- $X.\mathsf{read}_T()$ is invoked by the transaction T to read the base object X. That operation returns a value of X or the control value $abort$. If $abort$ is returned, the invoking transaction is aborted.
- $X.\mathsf{write}_T(v)$ 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 (when we do not consider the early abort mecanism).
- If a transaction attains its last statement (as defined by the user) it executes $\mathsf{try\_to\_commit}_T()$. That operation decides the fate of T by returning $commit$ or $abort$. (Let us notice, a transaction T that invokes $\mathsf{try\_to\_commit}_T()$ has not been aborted during an invocation of $X.\mathsf{read}_T()$.)

4.2 The STM system variables

To implement the previous STM 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.

- A logical clock denoted $CLOCK$. This clock, initialized to 0, can be read, and atomically increased with the $Fetch\&Increment()$ operation.
- A lock per base object X. Locks are assumed to be fair (assuming each lock is eventually released, every transaction that requires a lock is eventually granted the lock).
- 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.
- Each base object X is made up of two fields. The field $X.value$ denotes its current value, while the field $X.date$ 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. That variable allows the transaction T to safely evaluate the abstract property $P2(T)$. As we will see, we have $P2(T) \Rightarrow (MAX\_DATE_T = +\infty)$, and the STM system will direct T to commit when $MAX\_DATE_T = +\infty$ (Lemma 2 in Section 5.2).

In addition to accessing the previous variables kept in the shared memory, a transaction T manages the following local variables. The local control variables are denoted with lowercase letters.

- lrs_T and lrw_T are sets where T keeps the ids of the objects it has read and written, respectively.
- $read\_only_T$ is a boolean, initialized to $true$, that is set to $false$, if T invokes a $X.\mathsf{write}_T(v)$ operation.

- For each object X is accesses, T keeps a copy lcx in its local memory. Its two fields are denoted $lcx.value$ and $lcx.date$.
- min_date_T contains the greatest date of the objects T has read so far. Its initial value is 0. Combined with MAX_DATE_T, that variable allows a safe evaluation of the abstract property $P1(T)$. As we will see, we have $P1(T) \Rightarrow (min\_date_T \leq MAX\_DATE_T)$, and the STM system will not abort a read-only transaction T if $min\_date_T \leq MAX\_DATE_T$ (Lemma 3 in Section 5.2).

4.3 The algorithms of the STM system

The three operations that constitute the STM system $X.\mathsf{read}_T()$, $X.\mathsf{write}_T(v)$, and $\mathsf{try\_to\_commit}_T()$, are described in Figure 1. As in a lot of other protocols (e.g., STM or discrete event simulation), the underlying idea is to associate a time window, namely $[min\_date_T, MAX\_DATE_T]$, with each transaction T. This time window is managed as follows:

- When a read-only or update transaction T reads a new object (from the shared memory), it accordingly updates min_date_T, and aborts if its time window becomes empty. A time window becomes empty when the system is unable to guarantee that the values previously read by T and the value it has just obtained belong to a consistent snapshot.
- When an update transaction T is about to commit, it has two things to do. First, write into the shared memory the new values of the objects it has updated, and define their dates as the current clock value. It is possible that these writes make inconsistent the snapshot of a transaction T' that has already obtained values and will read a new object in the future. Hence, in order to prevent such an inconsistency from occurring (see the previous item), the transaction T sets $MAX\_DATE_{T'}$ to the current clock value if $\big((T' \in RS_X) \wedge (X \in lws_T)\big)$ and $(MAX\_DATE_{T'} = +\infty)$.

The operation $X.\mathsf{read}_T()$ When T invokes $X.\mathsf{read}_T()$, it obtains the value of X currently kept in the local memory if there is one (lines 01 and 08). Otherwise, T first allocates space in its local memory for a copy of X (line 02), obtains the value of X from the shared memory and updates RS_X accordingly (line 03). 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 read/write conflicts are announced in the proposed STM algorithm.

Then, T updates its local control variables lrs_T (line 04) and min_date_T (line 05) in order to keep them consistent. Finally, T checks its time window (line 06) 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.

Remark. Looking into the details, when a transaction T reads X from the shared memory, two causes can make true the window predicate $(min\_date_T > MAX\_DATE_T)$: min_date_T has just been increased, or MAX_DATE_T has been decreased to a finite value (or both). If the abort is due to an increase of min_date_T, T is aborted due to a write/read conflict on X. Differently, an abort caused by the fact that MAX_DATE_T has been set to a finite value, is due to a read/write conflict on $Y \neq X$.

The operation $X.\mathsf{write}_T()$ The text of the algorithm implementing the operation $X.\mathsf{write}_T()$ is very simple. The transaction first sets a flag to record that it is not a read-only transaction (line 09). If there is no local copy of X, 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. Finally, T updates the local copy of X (line 11), and records that it has locally written the copy of X (line 12). It is important to notice that an invocation of $X.\mathsf{write}_T()$

```
operation X.read_T():
(01) if (there is no local copy of X) then
(02)    allocate local space lcx for a copy;
(03)    lock X; lcx ← X; RS_X ← RS_X ∪ {T}; unlock X;
(04)    lrs_T ← lrs_T ∪ {X};
(05)    min_date_T ← max(min_date_T, lcx.date);
(06)    if (min_date_T > MAX_DATE_T) then return(abort) end if
(07) end if;
(08) return (lcx.value)
================================================================================
operation X.write_T(v):
(09) read_only_T ← false;
(10) if (there is no local copy of X) then allocate local space lcx for a copy end if;
(11) lcx.value ← v;
(12) lws_T ← lws_T ∪ {X}
================================================================================
operation try_to_commit_T():
(13) if (read_only_T)
(14)    then return(commit)
(15)    else lock all the objects in lrs_T ∪ lws_T;
(16)         if (MAX_DATE_T ≠ +∞) then release all the locks; return(abort) end if;
(17)         current_time ← CLOCK;
(18)         for each T' ∈ ( ∪_{X∈lws_T} RS_X ) do C&S(MAX_DATE_T', +∞, current_time) end for;
(19)         commit_time ← Fetch&Increment(CLOCK);
(20)         for each X ∈ lws_T do X ← (lcx.value, commit_time); RS_X ← ∅ end for;
(21)         release all the locks;
(22)         return(commit)
(23) end if
```

Figure 1: A clock+locks-based STM system

is purely local: it involves no access to the shared memory, and cannot entail an immediate abort of the corresponding transaction.

The operation $\mathsf{try\_to\_commit}_T()$ This operation works as follows. If the invoking transaction is a read-only transaction, it is committed (lines 13-14). So, a read-only transaction can abort only during the invocation of a $X.\mathsf{read}_T()$ operation (line 06 of that operation).

If the transaction T is an update transaction, $\mathsf{try\_to\_commit}_T()$ first locks all the objects accessed by T (line 15). (In order to prevent deadlocks, it is assumed that these objects are locked according to a predefined total order, e.g., their identity order.) Then, T checks if $MAX\_DATE_T \neq +\infty$. If this is the case, there is a read/write conflict: T has read an object that since then has been overwritten. Consequently, there is no guarantee for the current snapshot of T (that is consistent) and the write operations of T to appear as being atomic. T consequently aborts (after having released all the locks it has previously acquired, line 16).

If the predicate $MAX\_DATE_T = +\infty$ is true, T will necessarily commit. But, before releasing the locks and committing (lines 21-22), T has to (1) write in the shared memory the new values of the objects with their new dates (lines 19-20), and (2) update the control variables to indicate possible (read/write with read in the past, or write/read with read in the future) conflicts due to the objects it has written. As indicated at the beginning of this section, (1) read/write conflicts are managed by setting $MAX\_DATE_{T'}$ to the current clock value for all the transactions T' such that $\big((T' \in RS_X) \wedge (X \in lws_T)\big)$ (lines 17-18), and consequently RS_X is reset to $\emptyset$ (line 20), while (2) write/read conflicts on an object X are managed by setting the date of X to the commit time of T.

As two transactions $T1$ and $T2$ can simultaneously find $MAX\_DATE_{T'} = +\infty$ and try to change its value, the modification of $MAX\_DATE_{T'}$ is controlled by an atomic compare&swap operation (denoted $C\&S()$, line 18).

Remark 1. In order to save (expensive) $C\&S(MAX\_DATE_{T'}, +\infty, current\_time)$ at line 18, this invocation can be replaced by the following statement:

"**if** $(MAX\_DATE_{T'} = +\infty)$ **then** $C\&S(MAX\_DATE_{T'}, +\infty, commit\_time)$ **end if**".

Remark 2. It is worth noticing that the proposed algorithm does not address write/write conflicts. As we will see, the write-only transactions are never aborted.

4.4 Reducing the aborts

The predicate used at line 06 can be satisfied for $MAX\_DATE_T = d$ (and then T is aborted), while it would be false for $MAX\_DATE_T = d + 1$ (or a greater value). This means that, when it is updated to a finite value, MAX_DATE_T has to be set to a value as great as possible. On another side, $CLOCK$ can be increased by an arbitrary number of transactions between two successive accesses to $CLOCK$ by the transaction T (at line 17 and line 19).

If the aim is to abort as few transactions as possible (without adding other control variables)[2], a best effort strategy can be obtained by exploiting the previous observations. More precisely, replacing the lines 17 and 18 by the following statement

"**for each** $T' \in \big(\cup_{X \in lws_T} RS_X\big)$ **do** $C\&S(MAX\_DATE_{T'}, +\infty, CLOCK)$ **end for**"

can reduce the number of aborts. It is important to notice that a price has to be paid for this improvement: each $C\&S()$ invocation now requires an additional access to the shared memory to obtain the last value of $CLOCK$.

4.5 Favoring early abort

As indicated in the introduction, as soon as the fate of a transaction is to abort, it has to be aborted as soon as possible. In the proposed algorithm, the fate of a transaction T is to abort as soon as the predicate

$$(min\_date_T > MAX\_DATE_T) \ \vee \ \big(\neg read\_only_T \wedge (MAX\_DATE_T \neq +\infty)\big)$$

becomes true. Consequently, in order to expedite aborts, it is possible to:

- Add the statement "**if** $(min\_date_T > MAX\_DATE_T) \vee \big(\neg read\_only_T \wedge (MAX\_DATE_T \neq +\infty)\big)$ **then** return $(abort)$ **end if**" before line 01,
- Replace the statement of line 06 by the statement used in the previous item,
- Add the statement "**if** $(MAX\_DATE_T \neq +\infty)$ **then** return $(abort)$ **end if**" before line 09,
- And add the previous statement at line 15, just before locking the locks.

This is obtained at the additional price of increasing the number of shared memory accesses to the atomic variables MAX_DATE_T. (It is worth noticing that these predicates could easily be used by an underlying contention manager.)

[2] It is important to notice that a transaction T that entails the abort of another transaction T' does it by setting its variable $MAX\_DATE_{T'}$ to a finite value. But this can only occur at line 18, i.e., when T commits. Said another way, a transaction that aborts cannot entail the abort of another transaction.

5 Proof of the protocol

This section shows that the STM algorithm described in the previous section satisfies the specification stated in Section 3.3, namely termination, opacity and obligation.

5.1 Proof of the termination property

Theorem 3 *Let us assume that only a finite number of transactions can start during a finite period of time. Then, for each transaction T, if T terminates in a concurrency-free context, then T always terminates.*

Proof Considering a transaction T, the only operation of T that (from a liveness point of view) depends on the other transactions are the lock acquisitions (lines 03 and 15). As, by assumption, only a finite number of transactions start during a finite period of time, it follows that only a finite number of transactions can wait on a lock at any given time. Moreover, locks are fair and are held by processes for only a finite number of their own processing steps (during the $\mathsf{read}_T()$ and $\mathsf{try\_to\_commit}_T()$ operations). It follows from these observations that the transaction T is never blocked forever when waiting for a lock. Consequently, if T terminates when executed alone, it also terminates in concurrency context. $\square_{Theorem\ 3}$

Corollary 1 *Any transaction either commits or aborts.*

Proof The proof is an immediate consequence of Theorem 3, and the fact that a transaction invokes exactly once either $\mathsf{return}(commit)$ or $\mathsf{return}(abort)$. $\square_{Corollary\ 1}$

5.2 Proof of the obligation property

Let us consider a complete transaction history, i.e., a history in which each transaction has terminated. Due to Corollary 1, the sets $\mathcal{C}$ and $\mathcal{A}$ define a partition of the transactions. To prove the obligation property, we consider instead its contrapositive, namely $\forall\, T : \big(T \in \mathcal{A}\big) \Rightarrow \big(\neg P1(T) \vee \neg read\_only(T)\big) \wedge \neg P2(T)$.

Lemma 2 $\forall\, T : \big(T \in \mathcal{A}\big) \Rightarrow \neg P2(T)$.

Proof As indicated by the predicates used at line 06 or line 16, a transaction T can be aborted only if MAX_DATE_T is finite. This observation is the start of the following derivation:

$$\begin{array}{rcl} \big(T \in \mathcal{A}\big) & \Rightarrow & \text{at time } E_T,\ \ MAX\_DATE_T \neq +\infty, \\ (\text{Due to line 18}) & \Rightarrow & \exists T1, \exists X : \mathit{conflict}(X, R_T, W_{T1}) \wedge B_{T1} <_H E_T, \\ (\text{By definition of } P2(T)) & \Rightarrow & \neg P2(T). \end{array}$$

$\square_{Lemma\ 2}$

Lemma 3 $\forall\, T : \big(T \in \mathcal{A} \wedge read\_only(T)\big) \Rightarrow \neg P1(T)$.

Proof As indicated by the predicate of line 06, a read only transaction T can be aborted only if $min\_date_T > MAX\_DATE_T$. This leads to the following sequence of deductions:

$$\begin{array}{rcl}
\big(T \in \mathcal{A} \wedge read_only(T)\big) & \Rightarrow & \exists t : min_date_T > MAX_DATE_T \text{ at time } t <_H E_T, \\
\text{(Due to line 18)} & \Rightarrow & \exists T1, \exists X1 : conflict(X1, R_T, W_{T1}) \wedge MAX_DATE_T = current_time_{T1}, \\
\text{(and to lines 05 and 20)} & & \wedge\ \exists T2, \exists X2 : conflict(X2, W_{T2}, R_T) \wedge min_date_T = commit_time_{T2}, \\
& \Rightarrow & current_time_{T1} < commit_time_{T2}, \\
& \Rightarrow & B_{T1} <_H E_{T2}, \\
\text{(By definition of } P1(T)) & \Rightarrow & \neg P1(T).
\end{array}$$

$\square_{Lemma\ 3}$

Theorem 4 $\forall\, T : \big(P1(T) \wedge read\_only(T)\big) \vee P2(T) \Rightarrow \big(T \in \mathcal{C}\big)$.

Proof Immediate consequence of the Lemmas 2 and 3. $\square_{Theorem\ 4}$

Corollary 2 *Let T be a transaction. If (1) T is a read-only transaction and $P1(T)$ is satisfied, or (2) T is an update transaction and $P2(T)$ is satisfied, or (3) T is a write-only transaction, then T commits.*

Proof The items (1) and (2) are simple re-statements of Theorem 4. The item (3) follows from item (2) (a write-only transaction is also an update transaction) and the fact that $P2(T)$ is then trivially satisfied as a write-only transaction does not read objects. $\square_{Corollary\ 2}$

5.3 Proof of the opacity property

5.3.1 Additional definitions

- Let $AL_T(X, op)$ denote the event associated with the acquisition of the lock on the object X issued by the transaction T during an invocation of op where op is $X.\mathsf{read}_T()$ or $\mathsf{try\_to\_commit}_T()$. Similarly, let $RL_T(X, op)$ denote the event associated with the release of the lock on the object X issued by the transaction T during an invocation of op. Let us recall that, as $<_H$ (the shared memory history) is a total order, each event in H (including now $AL_T(X, op)$ and $RL_T(X, op)$) can be seen as a date of the time line. This "date" view of a sequential history on events will be used in the following proofs.
- The *read-from* relation between transactions, denoted $\rightarrow_{rf}$, is defined as follows: $T1 \stackrel{X}{\rightarrow}_{rf} T2$ if $T2$ reads the value that $T1$ wrote in the object X.

5.3.2 Principle of the proof of the opacity property

According to the algorithms implementing the operations $X.\mathsf{read}_T()$ and $X.\mathsf{write}_T(v)$ described in Figure 1, we ignore all the read operations on an object that follow another operation on the same object within the same transaction, and all the write operations that follow another write operation on the same object within the same transaction (these are operations local to the memory of the process that executes them). Building $\rho(TR)$ from TR is then a straightforward process.

To prove that the protocol described in Figure 1 satisfies the opacity consistency criterion, we need to prove that, for any transaction history $\widehat{TR}$ produced by this protocol, there is a sequential legal history $\widehat{ST}$ equivalent to $\widehat{\rho(TR)}$. This amounts to prove the following properties (where $\widehat{H}$ is the shared memory level history generated by the transaction history $\widehat{TR}$):

1. $\rightarrow_{ST}$ is a total order,
2. $\forall T \in TR : \big(T \text{ commits} \Rightarrow T \in ST\big) \wedge \big(T \text{ aborts} \Rightarrow \rho(T) \in ST\big)$,
3. $\rightarrow_{\rho(TR)} \subseteq \rightarrow_{ST}$,
4. $T1 \stackrel{X}{\rightarrow}_{rf} T2 \Rightarrow \nexists T3$ such that $\big(T1 \rightarrow_{ST} T3 \rightarrow_{ST} T2\big) \wedge \big(w_{T3}(X) \in H\big)$,
5. $T1 \stackrel{X}{\rightarrow}_{rf} T2 \Rightarrow T1 \rightarrow_{ST} T2$.

5.3.3 Definition of the linearization points

ST is produced by ordering the transactions according to their linearization points. The linearization point of the transaction T is denoted ℓ_T. The linearization points of the transactions are defined as follows :

- If a transaction T aborts, ℓ_T is the time at which its MAX_DATE_T global variable is assigned a finite value by a transaction T' (line 18 of the try_to_commit() operation of T').
- If a read-only transaction T commits, ℓ_T is placed at the earliest of (1) the occurrence time of the test during its last read operation (line 06 of the X.read() operation) and (2) the time at which MAX_DATE_T is assigned a finite value by another transaction. This value is unique and well-defined (this follows from the invocation of $C\&S(MAX\_DATE_{T'}, +\infty, current\_time)$ at line 18).
- If an update transaction T commits, ℓ_T is placed at the execution of line 19 by T (read and increase of the clock).

The total order $<_H$ (defined on the events generated by $\widehat{TR}$) can be extended with these linearization points. Transactions whose linearization points happen at the same time are ordered arbitrarily.

5.3.4 Proof of the opacity property

Let $\widehat{TR} = (TR, \rightarrow_{TR})$ be a transaction history. Let $\widehat{ST} = (\rho(TR), \rightarrow_{ST})$ be a history whose transactions are the transactions $\rho(TR)$, and such that $\rightarrow_{ST}$ is defined according to the linearization points of each transaction in $\rho(TR)$. If two transactions have the same linearization point, they are ordered arbitrarily. Finally, let us observe that the linearization points can be trivially added to the sequential history $\widehat{H} = (H, <_H)$ defined on the events generated by the transaction history $\widehat{TR}$. So, we consider in the following that the set H includes the transaction linearization points.

Lemma 4 *$\rightarrow_{ST}$ is a total order.*

Proof Trivial from the definition of the linearization points. $\square_{Lemma\ 4}$

Lemma 5 $\rightarrow_{\rho(TR)} \subseteq \rightarrow_{ST}$.

Proof This lemma follows from the fact that, given any transaction T, its linearization point is placed between its B_T and E_T events (that define its lifetime). Therefore, if $T1 \rightarrow_{\rho(TR)} T2$ ($T1$ ends before $T2$ begins), then $T1 \rightarrow_{ST} T2$. $\square_{Lemma\ 5}$

Let $finite(T, t)$ be the predicate "at time t, $MAX\_DATE_T \neq +\infty$".

Lemma 6 $finite(T, t) \Rightarrow \ell_T <_H t$.

Proof The proof of the lemma consists in showing that the linearization point of a transaction T cannot be after the time at which MAX_DATE_T is assigned a finite value. There are three cases.

- By construction, if T aborts, its linearization point ℓ_T is the time at which MAX_DATE_T is assigned a finite value, which proves the lemma.
- If T is read-only and commits, again by construction, its linearization point ℓ_T is placed at the latest at the time at which MAX_DATE_T is assigned a finite value (if it ever is), which again proves the lemma.
- If T writes and commits, its linearization point ℓ_T is placed during its try_to_commit() operation, while T holds the locks of every object that it has read. (If MAX_DATE_T had a finite value before it acquired all the locks, it would not commit due to line 16.) Let us notice that MAX_DATE_T can be assigned a finite value only by an update transaction holding a lock on a base object previously read by T. As T releases the locks just before committing (line 21), it follows that ℓ_T occurs before the time at which MAX_DATE_T is assigned a finite value, which proves the last case of the lemma.

$\square_{Lemma\ 6}$

Let $rs_X(T,t)$ be the predicate "at time t, T belongs to RS_X or $MAX\_DATE_T \neq +\infty$ ".

Lemma 7 $(T_W \xrightarrow{X}_{rf} T_R) \Rightarrow \nexists T'_W$ *such that* $\big(T_W \rightarrow_{ST} T'_W \rightarrow_{ST} T_R\big) \wedge \big(w_{T'_W}(X) \in H\big)$.

Proof The proof is by contradiction. Let us assume that there are transactions T_W, T'_W and T_R and an object X such that:

- $T_W \xrightarrow{X}_{rf} T_R$,
- $w_{T'_W}(X)v' \in H$,
- $T_W \rightarrow_{ST} T'_W \rightarrow_{ST} T_R$.

As both T_W and $T_{W'}$ write X (shared memory accesses), they have necessarily committed (a write in shared memory occurs only at line 20 during the execution of try_to_commit(), abbreviated ttc in the following). Moreover, their linearization points ℓ_{T_W} and $\ell_{T'_W}$ occur while they hold the lock on X (before committing), from which we have the following implications:

$$\begin{aligned}
T_W \rightarrow_{ST} T'_W &\Leftrightarrow \ell_{T_W} <_H \ell_{T'_W},\\
\ell_{T_W} <_H \ell_{T'_W} &\Rightarrow RL_{T_W}(X,\mathsf{ttc}) <_H AL_{T'_W}(X,\mathsf{ttc}),\\
RL_{T_W}(X,\mathsf{ttc}) <_H AL_{T'_W}(X,\mathsf{ttc}) &\Rightarrow w_{T_W}(X)v <_H w_{T'_W}(X)v',\\
\big(T_W \xrightarrow{X}_{rf} T_R\big) \wedge \big(w_{T_W}(X)v <_H w_{T'_W}(X)v'\big) &\Rightarrow w_{T_W}(X)v <_H r_{T_R}(X)v <_H w_{T'_W}(X)v'.
\end{aligned}$$

Hence, we have $(T_W \rightarrow_{ST} T'_W) \Rightarrow (r_{T_R}(X)v <_H w_{T'_W}(X)v')$.

On another side, a transaction T that reads an object X always adds its id to RS_X before releasing the lock on X. Therefore, the predicate $rs_X(T, RL_T(X, X.\mathsf{read}_T()))$ is true (a transaction T is removed from RS_X only after MAX_DATE_T has been assigned a finite value). Using this observation and the previous result, we have the following:

$$\begin{aligned}
r_{T_R}(X)v <_H w_{T'_W}(X)v' \wedge rs_X(T_R, RL_{T_R}(X, X.\mathsf{read}_{T_R}())) &\Rightarrow rs_X(T_R, AL_{T'_W}(X,\mathsf{ttc})),\\
\text{(Due to line 18)}\quad rs_X(T_R, AL_{T'_W}(X,\mathsf{ttc})) \wedge \big(w_{T'_W}(X)v' \in H\big) &\Rightarrow \mathit{finite}(T_R, \ell_{T'_W}),\\
\text{(Due to Lemma 6)}\quad \mathit{finite}(T_R, \ell_{T'_W}) &\Rightarrow \ell_{T_R} <_H \ell_{T'_W},\\
\text{(and finally)}\quad \ell_{T_R} <_H \ell_{T'_W} &\Leftrightarrow T_R \rightarrow_{ST} T'_W,
\end{aligned}$$

which proves that, contrarily to the initial assumption, T'_W cannot precede T_R in the sequential transaction history $\widehat{ST}$. $\square_{Lemma\ 7}$

Lemma 8 $(T_W \xrightarrow{X}_{rf} T_R) \Rightarrow (T_W \rightarrow_{ST} T_R)$.

Proof The proof is made up of two parts. First it is shown that $(T_W \xrightarrow{X}_{rf} T_R) \Rightarrow \neg \mathit{finite}(T_R, \ell_{T_W})$, and then it is shown that $\neg \mathit{finite}(T_R, \ell_{T_W}) \wedge T_W \xrightarrow{X}_{rf} T_R \Rightarrow (T_W \rightarrow_{ST} T_R)$.
Part 1: Proof of $(T_W \xrightarrow{X}_{rf} T_R) \Rightarrow \neg \mathit{finite}(T_R, \ell_{T_W})$.

Let us assume by contradiction that $\mathit{finite}(T_R, \ell_{T_W})$ is true. Due to the atomic $C\&S()$ operation used at line 18, $MAX\_DATE_{T_R}$ is assigned a finite value only once. $MAX\_DATE_{T_R}$ will then be strictly smaller than the value of $X.date$ after T_W writes it. The test at line 06 of the $X.\mathsf{read}_T()$ operation will then fail, leading to $\neg(T_W \xrightarrow{X}_{rf} T_R)$. Summarizing this reasoning, we have $\mathit{finite}(T_R, \ell_{T_W}) \Rightarrow \neg(T_W \xrightarrow{X}_{rf} T_R)$, whose contrapositive is what we wanted to prove.
Part 2: Proof of $\neg \mathit{finite}(T_R, \ell_{T_W}) \wedge T_W \xrightarrow{X}_{rf} T_R \Rightarrow (T_W \rightarrow_{ST} T_R)$.

As defined in Section 5.3.3, the linearization point ℓ_{T_R} depends on the fact that T_R commits or aborts, and is a read-only or update transaction. The proof considers the three possible cases.

- If T_R is an update transaction that commits, its linearization point ℓ_{T_R} occurs after its invocation of $\mathsf{try\_to\_commit}()$. Due to this observation, the fact that T_W releases its locks after its linearization point, and $T_W \xrightarrow{X}_{rf} T_R$, we have $\ell_{T_W} <_H \ell_{T_R}$, i.e., $T_W \rightarrow_{ST} T_R$.
- If T_R is a (read-only or update) transaction that aborts, its linearization point ℓ_{T_R} is the time at which $MAX\_DATE_{T_R}$ is assigned a finite value. Because $T_W \xrightarrow{X}_{rf} T_R$ we have $\neg \mathit{finite}(T_R, \ell_{T_W})$. Moreover, due to $\neg \mathit{finite}(T_R, \ell_{T_W})$ and the fact that T_R aborts, we have $\ell_{T_W} <_H \ell_{T_R}$, i.e., $T_W \rightarrow_{ST} T_R$. It follows that $T_W \xrightarrow{X}_{rf} T_R \Rightarrow T_W \rightarrow_{ST} T_R$.
- If T_R is a read-only transaction that commits, its linearization point ℓ_{T_R} is placed either at the time at which $MAX\_DATE_{T_R}$ is assigned a finite value (then the case is the same as a transaction that aborts, see before), or at the time of the test during its last read operation (line 06). In the latter case, we have $w_{T_W}(X)v <_H \ell_{T_W} <_H RL_{T_W}(X, \mathsf{ttc}) <_H AL_{T_R}(X, X.\mathsf{read}_{T_R}()) <_H r_{T_R}(X)v <_H \ell_{T_R}$, from which we have $\ell_{T_W} <_H \ell_{T_R}$, i.e., $T_W \rightarrow_{ST} T_R$.

Hence, in all cases, we have $(T_W \xrightarrow{X}_{rf} T_R) \Rightarrow (T_W \rightarrow_{ST} T_R)$. $\square_{Lemma\ 8}$

Theorem 5 *Every transaction history produced by the algorithm described in Figure 1 satisfies the opacity consistency property.*

Proof The proof follows from the construction of the set $\rho(TR)$ (Section 3.2, Section 5.3.2, and text of the algorithm), the definition of the linearization points (Section 5.3.3), and the Lemmas 4, 5, 7 and 8. $\square_{Theorem\ 5}$

6 Conclusion

This paper has presented two contributions in the context of software transactional memory. The first is the introduction and the statement of a provable property on transactions that obliges them to commit in "good"

circumstances. Such a property is a safety property that is a fundamental property for provably correct STM systems. Hence, it has been given the generic name *obligation property*. The second contribution is the design of a STM algorithm that implements the corresponding specification. This algorithm has been formally proved correct.

As noticed at the end of the introduction, the paper leaves open the following challenge: to find less constraining obligation properties (thereby forcing more transactions to commit) without requiring an algorithm implementing them to use "too many" additional control variables and remaining efficient when considering the ratio defined by the number of committed transactions divided by the total number of transactions.

References

[1] Avni H. and Shavit N., Maintaining Consistent Transactional States without a Global Clock. *Proc. 15th Colloquium on Structural Information and Communication Complexity (SIROCCO'08)*, To appear, 2008.

[2] 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.

[3] 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.

[4] Gramoli V., Harmanci D. and Felber P., Towards a Theory of Input Acceptance for Transactional Memories. *LPD-REPORT-2008-009*, Distributed Programming Lab, EPFL Lausanne (Switzerland), 2008.

[5] 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.

[6] 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.

[7] Imbs D. and Raynal M., A Lock-based Protocol for Software Transactional Memory. *Tech Report, #1893*, IRISA, Université de Rennes 1, France, May 2008.

[8] Lamport L., How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C28(9):690-691, 1979.

[9] Papadimitriou Ch.H., The Serializability of Concurrent Updates. *Journal of the ACM*, 26(4):631-653, 1979.

[10] Raynal M., Sequential Consistency as Lazy Linearizability. *BA. Proc. 14th ACM Symposium on Parallel Algorithms and Architectures (SPAA'02)*, pp. 151-152, Winnipeg, 2002.

[11] 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.

[12] Shavit N. and Touitou D., Software Transactional Memory. *Distributed Computing*, 10(2):99-116, 1997.