Multicore Programming — Projet
Mémoire transactionnelle logicielle

1 Introduction

En cours, nous avons vu deux façons génériques d'implémenter n'importe quel objet linéarisable. La première utilise des verrous : en empêchant physiquement des opérations de s'exécuter en concurrence, on est certain que la concurrence ne posera pas de problème. Le verrouillage à gros grain est certes très simple à mettre en œuvre mais réduit énormément le parallélisme, et donc les performances. Le verrouillage à grain fin est beaucoup plus difficile à mettre en œuvre. La seconde option se base sur le consensus ou des instructions spéciales, comme `compareAndSet`. Comme pour les solutions bloquantes, les constructions universelles non-bloquantes qui construisent un journal des opérations exécutées sont très inefficaces, et les algorithmes spécifiques à un objet donné, comme la pile, sont très complexes. Dans ce projet, nous explorons une troisième piste : les mémoires transactionnelles logicielles.

L'idée est la suivante : comme avec l'utilisation de verrous, le code à exécuter atomiquement doit être encadré par l'appel de deux fonctions, `begin` et `try_to_commit`. Entre les deux, il est possible de faire des lectures et des écritures sur des registres X en appelant les fonctions X.`read`() et X.`write`(v).

Les lectures, les écritures et l'invocation de la fonction `try_to_commit` peuvent *aborter*. Dans ce cas, on dit que la transaction complète aborte. Intuitivement, une transaction abortée n'a aucun effet et doit être redémarrée. Le critère de cohérence attendu sur les transactions est appelé *sérialisabilité*.Une exécution est sérialisable si elle est équivalente à une exécution séquentielle (c'est-à-dire dans laquelle il n'y a pas de concurrence entre les transactions) qui contient toutes (et ne contient que) les transactions qui n'ont pas aborté. Autrement dit, l'exécution dans laquelle toutes les transactions abortées ont été supprimées, est linéarisable.

La mémoire transactionnelle fournit les deux interfaces suivantes. Les fonctions `begin` et `try_to_commit` sont fournies comme des méthodes d'une classe implémentant `Transaction`. Cette interface fournit également une méthode `isCommited` retournant `true` si, et seulement si, `try_to_commit` a été appelée, s'est terminée sans lever d'exception, et `begin` n'a pas été appelée depuis. L'interface `Register<T>` fournit les méthodes permettant de lire et écrire dans un registre. Remarquez que les aborts sont ici gérés à l'aide d'exceptions.

```
interface Transaction {
  public void begin();
  public void try_to_commit() throws AbortException;
  public boolean isCommited();
}
interface Register<T> {
  public T read(Transaction t) throws AbortException;
  public void write(Transaction t, T v) throws AbortException;
}
```

Le code suivant présente un exemple d'utilisation de la mémoire transactionnelle. Il s'agit essentiellement d'une transaction contenant une lecture suivie d'une écriture, au sein d'une boucle réessayant d'exécuter la transaction jusqu'à ce qu'elle soit acceptée.

```
void increment (Register<Integer> X) {
  Transaction t = new STMTransaction();
  while (!t.isCommited()) {
    try {
      t.begin();
      X.write(t, X.read(t) + 1);
      t.try_to_commit();
    } catch (AbortException e) {}
  }
}
```

2 Algorithme *Transactional Locking 2*

Beaucoup d'algorithmes ont été proposés pour implémenter la mémoire transactionnelle, dont les plus simples sont expliqués par l'article de Raynal et Imbs [3]. La lecture de cet article, en annexe de ce sujet, est vivement recommandée. L'algorithme 1 présente Transactional Locking 2 (TL2). Il est basé sur une horloge logique globale, clock, incrémentée à chaque fois qu'une transaction est sur le point d'être acceptée (ligne 22), et qui décrit l'ordre (ou plutôt *un* ordre) dans laquelle les transactions acceptées sont sérialisées.

Chaque registre X accessible dans une transaction comporte deux champs : un champ de données X.`value` qui contient la dernière valeur écrite et un champ de contrôle X.`date` qui contient la date de la dernière transaction acceptée à avoir écrit dans X. De plus, un verrou est associé à chaque registre.

Chaque transaction T gère une variable locale lrs_T contenant l'ensemble des variables lues, une variable locale lws_T contenant l'ensemble des variables écrites au cours de T et une variable birthDate contenant la valeur de clock au début de la transaction.

Lors d'une écriture sur X, une copie locale de X est créée pour conserver la valeur écrite. De plus X est placée dans l'ensemble lws_T. Lors d'une lecture de X, si X a été précédemment écrite par la même transaction, la valeur locale est retournée. Sinon, X est placée dans l'ensemble lrs_T, puis il est vérifié que X n'a pas été modifiée depuis le début de la transaction en comparant la date de X à birthDate (auquel cas l'abort est nécessaire) et la valeur en mémoire partagée est retournée.

La fonction `try_to_commit` est bloquante. Lorsqu'elle est appelée, la transaction prend les verrous sur tous les registres puis vérifie que les valeurs lues sont toujours cohérentes entre elles, et cohérentes avec les nouvelles valeurs qu'elle a écrites localement. Cela est fait en comparant la date actuelle de chaque registre lu à la date birthDate du début de la transaction. Si l'une de ces dates est plus grande, un registre a été écrit par une transaction concurrente et un abort est nécessaire. Sinon, la transaction peut être acceptée : les écritures sont reportées sur la mémoire partagée, à une date obtenue en incrémentant l'horloge globale.

Algorithm 1: Algorithme TL2

```
operation $T$.begin()
    re-initialize local variables;
    birthDate $\leftarrow$ clock;
operation $X$.write($T, v$)
    if there is no local copy lcx of $X$ then
        allocate local space lcx for a copy
    lcx.value $\leftarrow v$; lws$_T \leftarrow$ lws$_T \cup \{X\}$;
operation $X$.read($T$)
    if there is a local copy lcx of $X$ then
        return lcx.value;
    else
        lcx $\leftarrow X$.copy; lrs$_T \leftarrow$ lrs$_T \cup \{X\}$;
        if lcx.date $>$ birthDate then abort;
        else return lcx.value;
operation $T$.try_to_commit()
    lock all the objects in lws$_T$;
    if some object in lrs$_T$ is already locked then
        release all the locks; abort;
    foreach $X \in$ lrs$_T$ do
        if $X$.date $>$ birthDate then
            release all the locks; abort;
    commitDate $\leftarrow$ clock.getAndIncrement();
    foreach $X \in$ lws$_T$ do
        $X \leftarrow$ (lcx.value, commitDate);
    release all the locks;
```

3 Travail demandé

Votre travail consiste à implémenter une mémoire transactionnelle logicielle fonctionnant selon l'algorithme TL2. Chaque fonction devra être starvation-free. Vous devrez rendre deux fichiers :

- un rapport succinct (au plus 4 pages) au format .pdf. Vous y discuterez en particulier de la propriété de vivacité sur les transactions garantie par l'algorithme TL2.
- une archive contenant votre code, y compris :
  - une implémentation de TL2,
  - une implémentation, avec des transactions, du dictionnaire décrit dans le TP 3, qui stocke également la pile des pages qu'il reste à explorer. Pouvez-vous compresser encore l'information nécessaire pour stocker l'ensemble des chaînes de caractères ?
  - Le code du TP 3 et l'implémentation de thread pool du distanciel mis à jour pour fonctionner avec l'objet décrit ci-dessus.

4 Références

Le concept de mémoire transactionnelle matérielle a été proposé par Herlihy et Moss en 1993 [1]. L'algorithme TL2 a été découvert par Dice, Shalev et Shavit en 2006 [2]. La présentation de l'algorithme et la formulation de ce sujet ont été largement inspirées d'un article de Imbs et Raynal de 2009 [3].

[1] Maurice Herlihy et J. Eliot B. Moss. *Transactional memory: architectural support for lock-free data structures.* International Symposium on Computer Architecture (1993)

[2] Dave Dice, Ori Shalev et Nir Shavit. *Transactional Locking II.* International Symposium on Distributed Computing (2006)

[3] Damien Imbs et Michel Raynal. *Software transactional memories: an approach for multicore programming.* International Conference on Parallel Computing Technologies (2009)

De nombreuses implémentations des mémoires transactionnelles logicielles sont disponibles en Java, même si aucune n'est universellement acceptée. Voici des liens vers quelques unes des plus connues.

AtomJava : `https://wasp.cs.washington.edu/wasp_atomjava.html`

JVSTM : `http://inesc-id-esw.github.io/jvstm`

Deuce STM : `https://sites.google.com/site/deucestm`

Multiverse : `https://github.com/pveentjer/Multiverse`

DSTM2 : `http://www.oracle.com/technetwork/indexes/downloads/index.html`

ObjectFabric : `https://github.com/cypof/objectfabric`

Étant données les limitations des mémoires transactionnelles logicielles, beaucoup de chercheurs militent pour l'introduction de mémoires transactionnelles matérielles, dans lesquelles les mêmes propriétés sont assurées par l'architecture physique de la mémoire partagée, au même titre que les caches.

Software Transactional Memories: An Approach for Multicore Programming

Damien Imbs and Michel Raynal

IRISA, Université de Rennes 1, 35042 Rennes, France
{damien.imbs,raynal}@irisa.fr

Abstract. The recent advance of multicore architectures and the deployment of multiprocessors as the mainstream computing platforms have given rise to a new concurrent programming impetus. Software transactional memories (STM) are one of the most promising approach to take up this challenge. The aim of a STM system is to discharge the application programmer from the management of synchronization when he/she has to write multiprocess programs. His/her task is to decompose his/her program in a set of sequential tasks that access shared objects, and to decompose each task in atomic units of computation. The management of the required synchronization is ensured by the associated STM system. This paper presents two STM systems, and a formal proof for the second one. Such a proof -that is not trivial- is one of the very first proofs of a STM system. In that sense, this paper strives to contribute to the establishment of theoretical foundations for STM systems.

Keywords: Concurrent programming, Consistent global state, Consistency condition, Linearizability, Lock, Logical clock, Opacity, Serializability, Shared object, Software transactional memory, Transaction.

1 Introduction

The challenging advent of multicore architectures. The speed of light has a limit. When combined with other physical and architectural demands, this physical constraint places limits on processor clocks: their speed is no longer rising. Hence, software performance can no longer be obtained by increasing CPU clock frequencies. To face this new challenge, (since a few years ago) manufacturers have investigated and are producing what they call *multicore architectures*, i.e., architectures in which each chip is made up of several processors that share a common memory. This constitutes what is called "the multicore revolution" [6].

The main challenge associated with multicore architectures is "how to exploit their power?" Of course, the old (classical) "multi-process programming" (multi-threading) methods are an answer to this question. Basically, these methods provide the programmers with the concept of a *lock*. According to the abstraction level considered, this lock can be a semaphore object, a monitor object, or the programmer's favorite synchronization object.

Unfortunately, traditional lock-based solutions have inherent drawbacks. On one side, if the set of data whose accesses are controlled by a single lock is too large

(large grain), the parallelism can be drastically reduced. On another side, the solutions where a lock is associated with each datum (fine grain), are error-prone (possible presence of subtle deadlocks), difficult to design, master and prove correct. In other words, providing the application programmers with locks is far from being the panacea when one has to produce correct and efficient multi-process (multi-thread) programs. Interestingly enough, multicore architectures have (in some sense) rang the revival of concurrent programming.

The Software Transactional Memory approach. The concept of *Software Transactional Memory* (STM) is an answer to the previous challenge. The notion of transactional memory has first been proposed (fifteen years ago) by Herlihy and Moss to implement concurrent data structures [7]. It has then been implemented in software by Shavit and Touitou [15], and has recently gained a great momentum as a promising alternative to locks in concurrent programming [3,5,12,14].

Transactional memory abstracts the complexity associated with concurrent accesses to shared data by replacing locking with atomic execution units. In that way, the programmer has to focus where atomicity is required and not on the way it has to be realized. The aim of a STM system is consequently to discharge the programmer from the direct management of synchronization entailed by accesses to concurrent objects.

More generally, STM is a middleware approach that provides the programmers with the *transaction* concept (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 shared 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).

Content of the paper. This paper is an introduction to STM systems, with an algorithmic and theoretical flavor. It is made up of three main sections. First, Section 2 presents a consistency condition suited to STM systems (called opacity [4]), and a STM architecture. Then, each of the two following sections presents a STM system that -in its own way- ensures the opacity consistency condition. The first (Section 3) is a simplified version of the well-known TL2 system [2] that has proved to be particularly efficient on meaningful benchmarks. The design of nearly all the STM systems proposed so far has been driven by efficiency, and nearly none of them has been proved correct. So, to give a broader view of the STM topic, the second STM system that is presented (Section 4) is formally proved correct[1]. Formal proofs are important as they provide STM systems with solid foundations, and consequently participate in establishing STM systems as a fundamental approach for concurrent programming [1].

2 A STM Computation Model

2.1 On STM Consistency Conditions

The classical consistency criterion for database transactions is serializability [13] (sometimes strengthened in "strict serializability", as implemented when using the 2-phase locking mechanism). The serializability consistency criterion involves only the transactions that are committed. Said differently, a transaction that aborts is not prevented from accessing an inconsistent state before aborting. In a STM system, the code encapsulated in a transaction can be any piece of code (involving shared data), it is not restricted to predefined patterns. Consequently a transaction always has to operate on a consistent state. To be more explicit, let us consider the following example where a transaction contains the statement $x \leftarrow a/(b-c)$ (where a, b and c are integer data), and let us assume that $b - c$ is different from 0 in all the consistent states. If the values of b and c read by a transaction come from different states, it is possible that the transaction obtains values such as $b = c$ (and $b = c$ defines an inconsistent state). If this occurs, the transaction raises an exception that has to be handled by the process that invoked the corresponding transaction[2]. Such bad behaviors have to be prevented in STM systems: whatever its fate (commit or abort) a transaction has to always see a consistent state of the data it accesses. The aborted transactions have to be harmless.

This is what is captured by the *opacity* consistency condition. Informally suggested in [2], and then formally introduced and deeply investigated in [4], opacity requires that no transaction reads values from an inconsistent global state. It is strict serializability when considering each committed transaction entirely, and an appropriate read prefix of each aborted transaction.

More precisely, let us associate with each aborted transaction T the read prefix that contains all its read operations until T aborts (if the abort is entailed by a read, this read is not included in the prefix). An execution of a set of transactions satisfies the *opacity* condition if all the committed transactions and the read prefix of each aborted transaction appear as if they have been executed one after the other, this sequential order being in agreement with their real time occurrence order. A formal definition of opacity appears in [4]. A very general framework for consistency conditions is introduced in [10], where is also introduced the *virtual world consistency* condition. A protocol implementing this general condition is described in [11].

[1] The STM system presented in this section has been introduced in [9] without being proved correct. So, this section validates and complements that paper.

[2] Even worse undesirable behaviors can be obtained when reading values from inconsistent states. This occurs for example when an inconsistent state provides a transaction with values that generate infinite loops.

2.2 The STM System Interface

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

- $\mathsf{begin}_T()$ is invoked by T when it starts. It initializes local control variables.
- $X.\mathsf{read}_T()$ 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 (in that case, the corresponding read does not belong to the read prefix associated with T).
- $X.\mathsf{write}_T(v)$ is invoked by the transaction T to update X to the new value v. That operation returns the control value ok or the control value $abort$. Like in the operation $X.\mathsf{read}_T()$, if $abort$ is returned, the invoking transaction is aborted.
- If a transaction attains its last statement (as defined by the user, which means it has not been aborted before) it executes the operation $\mathsf{try\_to\_commit}_T()$. 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 $X.\mathsf{read}_T()$.)

2.3 The Incremental Read/Deferred Update Model

In this transaction system model, each transaction T uses a local working space. When T invokes $X.\mathsf{read}_T()$ for the first time, it reads the value of X from the shared memory and copies it into its local working space. Later $X.\mathsf{read}_T()$ invocations (if any) use this copy. So, if T reads X and then Y, these reads are done incrementally, and the state of the shared memory can have changed in between.

When T invokes $X.\mathsf{write}_T(v)$, it writes v into its working space (and does not access the shared memory). Finally, if T is not aborted while it is executing $\mathsf{try\_to\_commit}_T()$, it copies the values written (if any) from its local working space to the shared memory. (A similar deferred update model is used in some database transaction systems.)

3 A Sketch of TL2 (Transactional Locking 2)

3.1 Aim and Principles

The TL2 STM system [2] aims at reducing the synchronization cost due to the read, write and validation (i.e., $\mathsf{try\_to\_commit}()$) operations. To that end, it associates a lock with each data object and uses a logical global clock (integer) that is read by all the transactions and increased by each writing transaction that commits. This global clock is basically used to validate the consistency of the state of the data objects a transaction is working on. The TL2 protocol is particularly efficient when there are few conflicts between concurrent transactions. (Two transactions conflict if they concurrently access the same object and one access is a write).

TL2 ensures the opacity property. The performance study depicted in [2] (based on a red-black tree benchmark) shows that TL2 is pretty efficient. It has nevertheless scenarios in which a transaction is directed to abort despite the fact that it has read values from a consistent state (these scenarios depend on the value of the global clock. They can occur when, despite the fact that all the values read by a transaction T are mutually consistent, one of them has been written by a transaction concurrent with T).

3.2 A Simplified Version of TL2

A very schematic description of the operations $\mathsf{begin}_T()$, $X.\mathsf{read}_T()$, $X.\mathsf{write}_T(v)$ and $\mathsf{try\_to\_commit}_T()$ used in TL2 [2] is given in Figure 1 for an update transaction. As indicated previously, the protocols implementing these operations are based on a global (logical) clock and a lock per object.

The global clock, denoted $CLOCK$, is increased (atomic $Fetch\&Increment()$ operation) each time an update transaction T invokes $\mathsf{try\_to\_commit}_T()$ (line 11). Moreover, when a transaction starts it invokes the additional operation $\mathsf{begin}_T()$ to obtain a birthdate (defined as the current value of the clock).

At the implementation level, an object X is made up of two fields: a data field $X.value$ containing the current value of the object, and a control field $X.date$ containing the date at which that value was created (line 12 of $\mathsf{try\_to\_commit}_T()$). A lock is associated with each object.

The case of an update transaction. Each transaction T manages a local read set lrs_T, and a local write set lws_T. As far as a $X.\mathsf{read}_T()$ is concerned we have the following. If, previously, a local copy lcx of the object X has been created by an invocation of $X.\mathsf{write}_T(v)$ issued by the same transaction, its value is returned (lines 01-02). Otherwise, X is read from the shared memory, and X's id is added to the local read set lrs_T (line 03). Finally, if the date associated with the current value of X is greater than the birthdate of T, the transaction is aborted (line 04). (This is because, T has possibly read other values that are no longer consistent with the value of X just obtained.) If the date associated with the current value of X is not greater than the birthdate of T, that value is returned by the $X.\mathsf{read}_T()$ operation. (In that case, the value read is consistent with the values previously read by T.)

The operation $X.\mathsf{write}_T(v)$ in TL2 and the one in the proposed protocol are similar. If there is no local copy of X, one is created and its value field is set to v. The local write set lws_T is also updated to remember that X has been written by T. The lifetime of the local copy lcx of X created by a $X.\mathsf{write}_T(v)$ operation spans the duration of the transaction T.

When a transaction T invokes $\mathsf{try\_to\_commit}_T()$ it first locks all the objects in $lrs_T \cup lws_T$. Then, T checks if the current values of the objects X it has read are still mutually consistent, and consistent with respect to the new values it has (locally) written. This is done by comparing the current date $X.date$ of each object X that has been read to the birthdate of T. If one of these dates is greater than its birthdate, there is a possible inconsistency and consequently T is aborted (line 11). Otherwise, T can be committed. Before being committed (line 14), T has to set the objects it has written to their new values (line 13). Their control part has also to be updated: they are set to the last clock value obtained by T (line 12). Finally, T releases the locks and commits.

Remark. This presentation of the $\mathsf{try\_to\_commit}_T()$ operation of TL2 does not take into account all of its aspects. As an example, if at line 09, all the locks cannot be

```
operation begin_T(): $birthdate \leftarrow CLOCK$.
==================================================================
operation $X$.read_T():
(01) if (there is a local copy $lcx$ of $X$)
(02)    then return ($lcx.value$) % the local copy $lcx$ was created by a write of $X$ %
(03)    else $lcx \leftarrow$ copy of $X$ read from the shared memory; $lrs_T \leftarrow lrs_T \cup \{X\}$;
(04)         if $lcx.date > birthdate$ then return ($abort$) else return ($lcx.value$) end if
(05) end if.
==================================================================
operation $X$.write_T($v$):
(06) if (there is no local copy of $X$) then allocate local space $lcx$ for a copy end if;
(07) $lcx.value \leftarrow v$; $lws_T \leftarrow lws_T \cup \{X\}$;
(08) return ($ok$).
==================================================================
operation try_to_commit_T():
(09)    lock all the objects in ($lrs_T \cup lws_T$);
(10)    for each $X \in lrs_T$ do % the date of $X$ is read from the shared memory %
(11)        if $X.date > birthdate$ then release all the locks; return ($abort$) end if end for;
(12)    $commit\_date \leftarrow Fetch\&Increment(CLOCK)$;
(13)    for each $X \in lws_T$ do $X \leftarrow (lcx.value, commit\_date)$ end for;
(14)    release all the locks; return ($commit$).
```

Fig. 1. TL2 algorithm for an update transaction

immediately obtained, TL2 can abort the transaction (and restart it later). This can allow for more efficient behaviors. Moreover, the lock of an object is used to contain its date value (this allows for more efficient read operations.)

The Case of a read only transaction. Such a transaction T does not modify the shared objects. The code of its X.read$_T$() and try_to_commit$_T$() operations can be simplified. This is left as an exercise for the reader.

4 A Window-Based STM System

This STM system has been proposed in [9] where is introduced the notion of ǒbligation property. It is called *window-based* because a logical time window is associated with each transaction, and a transaction has to be aborted when its window becomes empty. The opacity property does not prevent a STM system from aborting all the transactions. An obligation property states circumstances in which a STM system must commit a transaction T. Two obligation properties are defined in [9], and the aim of the system described in [9] is to satisfy both opacity and these obligation properties.

4.1 The STM Control Variables

The object fields, the object locks, the logical global clock, the local sets lrs_T and lws_T have the same meaning as in TL2. The following additional (shared or local) control variables are also used:

- 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.
- 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 $MAX\_DATE_T = +\infty$.
- $read\_only_T$ is a local boolean, initialized to $true$, that is set to *false*, if T invokes a $X.\mathsf{write}_T(v)$ operation.
- min_date_T is a local variable containing 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 $min\_date_T \leq MAX\_DATE_T$.

4.2 The STM Operations

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 2. 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. These writes may render 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 07). 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 and min_date_T (line 04) in order to keep them consistent. Finally, T checks its time window (line 05) 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, a single cause can cause the window predicate $(min\_date_T > MAX\_DATE_T)$ to be true: min_date_T has just been increased, and MAX_DATE_T has been decreased to a finite value. T is then aborted due to a write/read conflict on X and a read/write conflict on $Y \neq X$.

```
operation $X$.read$_T()$:
(01) if (there is no local copy of $X$) then
(02)    allocate local space $lcx$ for a copy;
(03)    lock $X$; $lcx \leftarrow X$; $RS_X \leftarrow RS_X \cup \{T\}$; unlock $X$;
(04)    $lrs_T \leftarrow lrs_T \cup \{X\}$; $min\_date_T \leftarrow \max(min\_date_T, lcx.date)$;
(05)    if $(min\_date_T > MAX\_DATE_T)$ then return$(abort)$ end if
(06) end if;
(07) return $(lcx.value)$.
=================================================================
operation $X$.write$_T(v)$:
(08) $read\_only_T \leftarrow false$;
(09) if (there is no local copy of $X$) then allocate local space $lcx$ for a copy end if;
(10) $lcx.value \leftarrow v$; $lws_T \leftarrow lws_T \cup \{X\}$;
(11) return $(ok)$.
=================================================================
operation try_to_commit$_T()$:
(12) if $(read\_only_T)$
(13)    then return$(commit)$
(14)    else lock all the objects in $lrs_T \cup lws_T$;
(15)         if $(MAX\_DATE_T \neq +\infty)$ then release all the locks; return$(abort)$ end if;
(16)         $current\_time \leftarrow CLOCK$;
(17)         for each $T' \in \big(\cup_{X \in lws_T} RS_X\big)$
                 do $C\&S(MAX\_DATE_{T'}, +\infty, current\_time)$ end for;
(18)         $commit\_time \leftarrow Fetch\&Increment(CLOCK)$;
(19)         for each $X \in lws_T$ do $X \leftarrow (lcx.value, commit\_time)$; $RS_X \leftarrow \emptyset$ end for;
(20)         release all the locks; return$(commit)$
(21) end if.
```

Fig. 2. A window-based STM system

The operation X.write$_T()$. The text of the algorithm implementing X.write$_T()$ is very simple. The transaction first sets a flag to record that it is not a read-only transaction (line 08). If there is no local copy of X, corresponding space is allocated in the local memory (line 09); let us remark that this does not entail a read of X from the shared memory. Finally, T updates the local copy of X, and records in lrw_T that it has locally written the copy of X (line 10). 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 $\mathsf{try\_to\_commit}_T()$. This operation works as follows. If the invoking transaction is a read-only transaction, it is committed (lines 12-13). So, a read-only transaction can abort only during the invocation of a $X.\mathsf{read}_T()$ operation (line 05).

If the transaction T is an update transaction, $\mathsf{try\_to\_commit}_T()$ first locks all the objects accessed by T (line 14). (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 15).

If the predicate $MAX\_DATE_T = +\infty$ is true, T will necessarily commit. But, before releasing the locks and committing (line 20), T has to (1) write in the shared memory the new values of the objects with their new dates (lines 18-19), 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 16-17), and consequently RS_X is reset to $\emptyset$ (line 19), 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 17).

4.3 Formal Framework to Prove the Opacity Property

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 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.

History at the shared memory level. 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.)

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*.

Sequential, equivalent and linearizable histories. 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 $\widehat{AA}$ [8]. If a transaction history $\widehat{AA}$ is linearizable it is possible to associate a single point of the time line with every transaction, no two transactions being associated with the same point. This point is called the *linearization point* of the corresponding transaction.

Reduced histories. Given a run of transactions $\widehat{TR} = \big(TR, \rightarrow_{TR}\big)$, let $\mathcal{C}$ (resp. $\mathcal{A}$) be the set of transactions that commit (resp., abort) in that run.

Given $T \in \mathcal{A}$, 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 $\mathit{prefix}(T)$ 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 $\mathit{prefix}(T)$ 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 $\rightarrow_{\rho(TR)} = \rightarrow_{TR}$. Opacity states that the transactions in $\mathcal{C} \cup \mathcal{A}'$ can be consistently and totally ordered according to their real-time order, i.e., $\widehat{\rho(TR)}$ is linearizable.

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: $\mathit{conflict}(X, W_{T1}, W_{T2}) \stackrel{\text{def}}{=} \big(w_{T1}(X) <_H w_{T2}(X)\big)$.

The read-from relation. 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.

4.4 A Formal Proof of the Opacity Property

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 2, 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 2 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. $T1 \stackrel{X}{\rightarrow}_{rf} T2 \Rightarrow T1 \rightarrow_{ST} T2$.

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 17 of the $\mathsf{try\_to\_commit}()$ 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 05 of the $X.\mathsf{read}()$ 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 $C\&S(MAX\_DATE_{T'}, +\infty, \mathit{current\_time})$ at line 17).
- If an update transaction T commits, ℓ_T is placed at the execution of line 18 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.

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 recall 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 1. *$\rightarrow_{ST}$ is a total order.*

Proof. Trivial from the definition of the linearization points. □

Lemma 2. $\rightarrow_{\rho(TR)} \subseteq \rightarrow_{ST}$.

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 $T1 \rightarrow_{ST} T2$. □

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

Lemma 3. *$\mathit{finite}(T, t) \Rightarrow \ell_T <_H t$.*

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 the control variable 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, 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 15.) 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 20), 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. □

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

In the following, $AL_T(X, op)$ denotes 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.read$_T$() or try_to_commit$_T$().

Similarly, $RL_T(X, op)$ denotes 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.

Lemma 4. $(T_W \overset{X}{\rightarrow}_{rf} T_R) \Rightarrow \nexists T'_W$ *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 (1) $T_W \overset{X}{\rightarrow}_{rf} T_R$, (2) $w_{T'_W}(X)v' \in H$ and (3) $T_W \rightarrow_{ST} T'_W \rightarrow_{ST} T_R$.

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 19 during the execution of $\mathsf{try\_to\_commit}()$, 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:

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

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). From this observation and the previous result, we have: $r_{T_R}(X)v <_H w_{T'_W}(X)v' \wedge rs_X(T_R, RL_{T_R}(X, X.\mathsf{read}_{T_R}())) \Rightarrow rs_X(T_R, AL_{T'_W}(X, \mathsf{ttc}))$, and then

$$\begin{aligned}
\text{(Due to line 17)} \quad rs_X(T_R, AL_{T'_W}(X, \mathsf{ttc})) \wedge \big(w_{T'_W}(X)v' \in H\big) &\Rightarrow \mathit{finite}(T_R, \ell_{T'_W}),\\
\text{(Due to Lemma 3)} \quad \mathit{finite}(T_R, \ell_{T'_W}) &\Rightarrow \ell_{T_R} <_H \ell_{T'_W},\\
\text{(and finally)} \quad \ell_{T_R} <_H \ell_{T'_W} &\Leftrightarrow T_R \rightarrow_{ST} T'_W,
\end{aligned}$$

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

Lemma 5. $(T_W \overset{X}{\rightarrow}_{rf} T_R) \Rightarrow (T_W \rightarrow_{ST} T_R)$.

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

Part 1: Proof of $(T_W \overset{X}{\rightarrow}_{rf} T_R) \Rightarrow \neg\mathit{finite}(T_R, \ell_{T_W})$.

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 17, $MAX\_DATE_{T_R}$ 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 05 of the $X.\mathsf{read}_T()$ operation will then fail, leading to $\neg(T_W \overset{X}{\rightarrow}_{rf} T_R)$. Summarizing this reasoning, we have $\mathit{finite}(T_R, \ell_{T_W}) \Rightarrow \neg(T_W \overset{X}{\rightarrow}_{rf} T_R)$, whose contrapositive is what we wanted to prove.

Part 2: Proof of $\neg\mathit{finite}(T_R, \ell_{T_W}) \wedge T_W \overset{X}{\rightarrow}_{rf} T_R \Rightarrow (T_W \rightarrow_{ST} T_R)$.

As defined earlier, 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 $T_W \overset{X}{\rightarrow}_{rf} T_R$, 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 $T_W \overset{X}{\rightarrow}_{rf} T_R$ 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 $T_W \overset{X}{\rightarrow}_{rf} T_R \Rightarrow T_W \rightarrow_{ST} T_R$.
- 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 05). 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 $(T_W \overset{X}{\rightarrow}_{rf} T_R) \Rightarrow (T_W \rightarrow_{ST} T_R)$. □

Theorem 1. *Every transaction history produced by the algorithm described in Figure 2 satisfies the opacity consistency property.*

Proof. The proof follows from the construction of the set $\rho(TR)$ (Section 4.3, Section 4.4, and text of the algorithm), the definition of the linearization points (Section 4.4), and the Lemmas 1, 2, 4 and 5. □

5 Conclusion

The aim of this paper was to show that Software Transactional Memory is a novel promising approach to address multiprocess programming. It discharges the programmer from using and managing base synchronization mechanism. The programmer only has to focus his/her attention (1) on the decomposition of his/her application into processes, and, for each process, (2) on its decomposition into atomic units.

To illustrate these notions, two STM protocols have been presented. Both are based on a logical global clock, and on locks associated with each shared object. The second protocol has been proved correct. Such a proof constitutes a step in establishing the foundations of STM systems.

References

1. Attiya, H.: Needed: Foundations for Transactional Memory. ACM Sigact News, Distributed Computing Column 39(1), 59–61 (2008)
2. Dice, D., Shalev, O., Shavit, N.: Transactional Locking II. In: Dolev, S. (ed.) DISC 2006. LNCS, vol. 4167, pp. 194–208. Springer, Heidelberg (2006)
3. Felber, P., Fetzer, Ch., Guerraoui, R., Harris, T.: Transactions are coming Back, but Are They The Same? ACM Sigact News, Distributed Computing Column 39(1), 48–58 (2008)
4. Guerraoui, R., Kapałka, M.: On the Correctness of Transactional Memory. In: Proc. 13th ACM SIGPLAN Symp. on Principles and Practice of Par. Progr. (PPoPP 2008), pp. 175–184 (2008)
5. Harris, T., Cristal, A., Unsal, O.S., Ayguade, E., Gagliardi, F., Smith, B., Valero, M.: Transactional Memory: an Overview. IEEE Micro 27(3), 8–29 (2007)
6. Herlihy, M.P., Luchangco, V.: Distributed Computing and the Multicore Revolution. ACM SIGACT News 39(1), 62–72 (2008)
7. Herlihy, M.P., Moss, J.E.B.: Transactional Memory: Architectural Support for Lock-free Data Structures. In: Proc. 20th ACM Int'l Symp. on Comp. Arch (ISCA 1993), pp. 289–300 (1993)
8. Herlihy, M.P., Wing, J.M.: Linearizability: a Correctness Condition for Concurrent Objects. ACM Transactions on Programming Languages and Systems 12(3), 463–492 (1990)
9. Imbs, D., Raynal, M.: Provable STM Properties: Leveraging Clock and Locks to Favor Commit and Early Abort. In: Garg, V., Wattenhofer, R., Kothapalli, K. (eds.) ICDCN 2009. LNCS, vol. 5408, pp. 67–78. Springer, Heidelberg (2008)
10. Imbs, D., Raynal, M.: On the Consistency Conditions of Transactional Memories. Tech Report #1917, 23 pages, IRISA, Université de Rennes, France (2009)
11. Imbs, D., Raynal, M.: A versatile STM protocol with invisible read operations that satisfies the virtual world consistency condition. In: 16th Colloquium on Structural Information and Communication Complexity (SIROCCO 2009). LNCS. Springer, Heidelberg (2009)
12. Larus, J., Kozyrakis, Ch.: Transactional Memory: Is TM the Answer for Improving Parallel Programming? Communications of the ACM 51(7), 80–89 (2008)
13. Papadimitriou, Ch.H.: The Serializability of Concurrent Updates. Journal of the ACM 26(4), 631–653 (1979)
14. Raynal, M.: Synchronization is coming back, but is it the same? Keynote Speech. In: IEEE 22nd Int'l Conf. on Advanced Inf. Networking and Applications (AINA 2008), pp. 1–10 (2008)
15. Shavit, N., Touitou, D.: Software Transactional Memory. Distributed Computing 10(2), 99–116 (1997)