Consistency for Transactional Memory Computing

Dmytro Dziuma[2], Panagiota Fatourou[1], and Eleni Kanellou[3]

[1] FORTH ICS & University of Crete, Heraklion (Crete), Greece
faturu@csd.uoc.gr
[2] FORTH ICS, Heraklion (Crete), Greece
dixond@acm.lviv.ua
[3] FORTH ICS, Heraklion (Crete), Greece & University of Rennes 1, Rennes, France
kanellou@ics.forth.gr

Abstract. This chapter provides *formal definitions* for a comprehensive collection of consistency conditions for transactional memory (TM) computing. We express all conditions in a uniform way using a formal framework that we present. For each of the conditions, we provide two versions: one that allows a transaction T to read the value of a data item written by another transaction T' that can be live and not yet commit-pending provided that T' will eventually commit, and a version which allows transactions to read values written only by transactions that have either committed before T starts or are commit-pending. Deriving the first version for a consistency condition was not an easy task but it has the benefit that this version is weaker than the second one and so it results in a wider universe of algorithms which there is no reason to exclude from being considered correct. The formalism for the presented consistency conditions is not based on any unrealistic assumptions, such as that transactional operations are executed atomically or that write operations write distinct values for data items. Making such assumptions facilitates the task of formally expressing the consistency conditions significantly, but results in formal presentations of them that are unrealistic, i.e. that cannot be used to characterize the correctness of most of the executions produced by any reasonable TM algorithm.

1 Introduction

Software Transactional memory (or STM for short) [21,35] is a promising programming paradigm that aims at simplifying parallel programming by using the notion of a transaction. A *transaction* executes a piece of code containing accesses to pieces of data, known as *data items*, which are accessed simultaneously by several threads in a concurrent setting. A transaction may either *commit* and then its updates take effect or *abort* and then its updates are discarded. By using transactions, the naive programmer needs only enhance its sequential code with invocations of special routines (which we call *transactional operations*, or *t-operations* for short) to read or write data items. When a transaction executes all its reads and writes on data items, it tries to commit. From that point on and until its completion, the transaction is *commit-pending*. Once a transaction starts and before its completion, it is *live*.

The STM algorithm provides implementations for t-operations (from base objects) so that all synchronization problems that may arise during the concurrent execution of

R. Guerraoui and P. Romano (Eds.): Transactional Memory, LNCS 8913, pp. 3–31, 2015.

transactions are addressed. The implementation details of the STM algorithm are hidden from the naive programmer whose programming task is therefore highly simplified. STM has been given special attention in the last ten years with hundreds of papers addressing different problems arising in STM computing (see e.g. [20,19] for books addressing different aspects of STM computing).

One of the most fundamental problems of STM computing is to define when an STM algorithm is correct. Most STM consistency conditions [4,18,19,24,15,9,10] originate from existing shared memory or database consistency models. However, in contrast to what happens in shared memory models where correctness has been defined in the granularity of single operations on shared objects, correctness in STM computing is defined in terms of *transactions*, each of which may invoke more than one read or write t-operations on data items. Comparing now to database transactions, the main difficulty when presenting consistency conditions for STM computing is that the execution of a t-operation has duration and is usually overlapping with the execution of other t-operations, whereas in database transactions reads and writes are considered to be atomic. For these reasons, existing consistency conditions for these two settings (shared memory and database concurrent transactions) cannot be applied verbatim to STM algorithms. Formalizing consistency conditions for STM computing requires more effort.

This chapter presents a comprehensive collection of consistency conditions for STM computing. All conditions are expressed in a uniform way using a formal framework that we present in Section 2. This chapter can therefore serve as a survey of *consistency conditions* for STM computing. However, it aspires to be more than this.

For all known STM consistency conditions we provide a new version, called *eager*, in which a transaction T is allowed to read the value of a data item written by another transaction T' that can be live and not yet commit-pending provided that T' will eventually commit (or that T' will commit if T commits). Most STM consistency conditions [4,9,10,18,19,24] presented thus far did not allow a transaction to read values that have been written by transactions that are neither committed nor commit-pending; we call this version of a consistency condition *deferred-update* (or *du* for short). The eager version of a consistency condition is weaker than its deferred-update version, thus resulting in a wider universe of algorithms which should not be excluded from being considered correct. For instance, in a database system, a transaction T may perform a *dirty read*, i.e. T may read a value v for a data item x written by a transaction T' which is still live (and not commit-pending) when T's read of x completes. To ensure the well-known consistency condition from databases, called recoverability [8], one technique described in the database literature [40], is to employ deferred commits and enforce cascading aborts whenever necessary. This is usually achieved by providing sufficient bookkeeping to determine essential orderings of commit and abort events that need to be enforced. In the aforementioned scenario, T has to defer its commit until T' completes its execution, and it necessarily aborts in case T' aborts. If an STM algorithm worked in a similar way, there would be no reason for executions of the algorithm not to be considered correct. However, current consistency conditions, as they are formally expressed, exclude such executions from the set of executions they allow. The eager version of a consistency condition we present here solves this problem.

In [37], Siek and Wojciechowski discuss why well-known STM consistency conditions, like opacity [18], serializability [29], virtual world consistency [24], and the TMS family [15] fail to support early release [30,36]. Early release is a technique introduced for optimizing performance; it allows a transaction to read a value for a data item written by another live transaction that is not commit-pending. Siek and Wojciechowski also discuss in [37] how one can design consistency conditions that support early release. They then use the proposed conditions to characterize the correctness of a distributed STM system they present in [36]. The way the eager versions of the consistency conditions are formulated in this chapter is flexible enough to support early release.

It is remarkable that deriving the eager version of consistency conditions was not an easy task so we consider their presentation as a significant contribution of this chapter. For the derivation of the presented consistency conditions, we do not make any restrictive assumptions, such as that t-operations are executed atomically or that writes write distinct values for data items. Making such assumptions is unrealistically restrictive since all STM algorithms produce executions that do not satisfy these assumptions. Thus, a consistency condition that has been expressed making such an assumption cannot be used to characterize such executions, and thus fail to also characterize whether the STM algorithm itself satisfies the condition. We remark that making such assumptions significantly facilitates the task of formally expressing a consistency condition but the formal presentation of the condition that results is very restrictive since it cannot be used to characterize the correctness of most of the executions produced by any reasonable STM algorithm.

Among the consistency conditions met in STM computing papers are strict serializability [29], serializability [29], opacity [18,19], virtual world consistency [24], TMS1 [15] (and TMS2 [15]), and snapshot isolation [3,13,32,9,10]. Weaker consistency conditions like processor consistency [10], causal serializability [9,10] and weak consistency [10] have also been considered in the STM context when proving impossibility results.

Strict serializability, as well as serializability, are usually presented in an informal way in STM papers which cite the original paper [29] where these conditions have first appeared in the context of database research. Thus, the differences that exist between database and STM transactions have been neglected in STM research. We present formal definitions of these consistency conditions here. Additional consistency conditions originating from the database research are presented in [4]. To present their formalism, the authors of [4] make the restrictive assumption that t-operations are atomic. The presentation of most of the other consistency conditions (e.g. opacity [18,19], virtual world consistency [24], snapshot isolation [3,13,32,9,10] and weaker variants of them [9,10]) is based on the assumption that a read for a data item by a transaction T can read a value written by either a transaction that has committed or is commit-pending when T starts its execution. Finally, the definition of virtual world consistency [24] is based on the assumption that each instance of WRITE writes a distinct value for the data item it accesses (or that the t-operations are executed atomically).

In this chapter, we do not cope with transactions whose code is determined at run time (i.e. after the beginning of the execution of the transaction). For instance, such a transaction could be produced on a web environment by deciding the next t-operations to be invoked by the transaction while executing it. We also do not discuss consistency issues that arise when data items are accessed not only by transactions but also outside the transactional scope (as it is e.g. the case for systems that support *privatization* [1,26,34,25,27,38]).

The rest of this chapter is organized as follows. Section 2 presents the formal framework which is employed in Section 3 to express the studied consistency conditions. Table 1 shows the relationships between consistency conditions.

2 Model

2.1 System

The system is asynchronous with a set of threads executed in it. Each thread is sequential (i.e. it executes a single sequential program) but different threads can be executed concurrently. Threads communicate via shared memory, i.e. by accessing simple shared objects, called base objects, usually provided by the hardware. Formally, a *base object* has a state and supports a set of operations, called *primitives*, to read or update its state. Base objects are usually as simple as read/write or `CAS` objects. A read/write object O stores a value from some set and supports two atomic primitives `read` and `write`; `read`(O) returns the current value of object O without changing it, and `write`(O, v) writes the value v into O and returns an acknowledgement. A `CAS` object O stores a value and supports, in addition to `read`, the atomic primitive `CAS`(O, v', v) which checks whether the value of O is v' and, if so, it sets the value of O to v and returns `true`, otherwise, it returns `false` and the value of O remains unchanged.

We model each thread as a state machine. A *configuration* describes the system at some point in time, so it provides information about the state of threads and the state of base objects. In an *initial configuration*, threads and base objects are in initial states. A *step* of a thread consists of applying a single primitive on some base object, the response to that primitive, and zero or more local computation performed by the thread; local computation accesses only local variables of the thread, so it may cause the internal state of the thread to change but it does not change the state of any base object. As a step, we will also consider the invocation of a routine or the response to such an invocation; notice that a step of this kind (1) is either the first or the last when executing the routine (more steps may be needed after the invocation of the routine in order for it to respond), and (2) does not change the state of any base object. Each step is executed atomically. An *execution* α is an alternating sequence of configurations and steps starting with an initial configuration. An execution is *legal* if the sequence of steps performed by each thread follows the algorithm for that thread and, for each base object, the responses to the primitives performed on the base object are in accordance with its specification (and the state of the base object at the configuration that the primitive is applied).

2.2 STM Definitions

Transactions and t-Operations. A *transaction* is a piece of sequential code which accesses (reads or writes) pieces of data, called *data items*. A data item may be accessed by several threads simultaneously when a transaction is executed in a concurrent environment. Transactions call specific routines, called READ and WRITE, to read and update, respectively, data items. A transaction may *commit* and then all its updates to data items take effect, or *abort* and then all its updates are discarded.

An STM algorithm uses a collection of base objects to store the state of data items. It also provides an implementation, for each thread, for READ and WRITE (from the base objects). READ receives as argument the data item x to be accessed (and possibly the thread p invoking READ and the transaction T for which p invokes READ) and returns either a value v for x or a special value A_T which identifies that T has to abort. WRITE receives as arguments the data item x to be modified, a value v (and possibly the thread p invoking WRITE and the transaction T for which p invokes WRITE), and returns either an acknowledgment or A_T. The STM algorithm provides implementations for two additional routines, called COMMIT and ABORT, which are called to try to commit or to abort a transaction, respectively. When COMMIT is executed by some transaction T it returns either a special value C_T, which identifies that T has committed, or A_T. ABORT always returns A_T.

We refer to all these routines as *t-operations*. A t-operation starts its execution when the thread executing it issues an *invocation* for it; the t-operation completes its execution when the thread executing it receives a *response*. Thus, the execution of a t-operation *op* is not atomic, i.e. the thread executing it may perform a sequence of primitives on base objects in order to complete the execution of the t-operation. Moreover, the invocation and the response of *op* are considered as two separate steps (with each of them being atomic). The invocation and the response of a t-operation are referred to as *events*. We sometimes say that these events are caused by T.

Histories. A *history* is a finite sequence of events. Consider any history H. A transaction T (executed by a thread p) *is in* H or H *contains* T, if there are invocations and responses of t-operations in H issued (or received) by p for T. The *transaction subhistory* of H for T, denoted by $H|T$, is the subsequence of all events in H issued by p for T. We say that a response *res matches* an invocation *inv* of a t-operation *op* in some history H, if they are both by the same thread p, *res* follows *inv* in H, *res* is a response for *op*, and there is no other event by p between *inv* and *res* in H. A history H is said to be *well-formed* if, for each transaction T in H, $H|T$ is an alternating sequence of invocations and matching responses, starting with an invocation, such that:

- no events in $H|T$ follow C_T or A_T;
- if T' is any transaction in H executed by the same thread that executes T, either the last event of $H|T$ precedes in H the first event of $H|T'$ or the last event of $H|T'$ precedes in H the first event of $H|T$.

From now on we focus on well-formed histories. Assume that H is such a history. A t-operation is *complete* in H, if there is a response for it in H; otherwise, the t-operation is *pending*. Thus, in H, there are two events for every complete t-operation *op*, an invocation $inv(op)$ and a matching response $res(op)$; moreover, H contains only one event for each pending t-operation in it, namely its invocation. A transaction T is *committed* in H, if $H|T$ includes C_T; a transaction T is *aborted* in H, if $H|T$ includes A_T. A transaction is *complete* in H, if it is either committed or aborted in H, otherwise

History H

p: T_1.READ(x)
p': T_2.READ(x)
p: $T_1.v$
p': $T_2.v$
T_2.READ(y)
p: T_1.WRITE(x, v')
$T_1.ok$
T_1.COMMIT
p': $T_2.v''$
p: $T_1.C_{T_1}$
T_3.WRITE(z, v)
p': T_2.COMMIT
$T_2.C_{T_2}$
p: $T_3.ok$

Subhistory $H|p$

p: T_1.READ(x)
$T_1.v$
T_1.WRITE(x, v')
$T_1.ok$
T_1.COMMIT
$T_1.C_{T_1}$
T_3.WRITE(z, v)
$T_3.ok$

Subhistory $H|T_2$

p': T_2.READ(x)
$T_2.v$
T_2.READ(y)
$T_2.v''$
T_2.COMMIT
$T_2.C_{T_2}$

History H'

p': T_2.READ(x)
p: T_1.READ(x)
p': $T_2.v$
T_2.READ(y)
p: $T_1.v$
p': $T_2.v''$
T_2.COMMIT
p: T_1.WRITE(x, v')
$T_1.ok$
T_1.COMMIT
p': $T_2.C_{T_2}$
p: $T_1.C_{T_1}$
T_3.WRITE(z, v)
$T_3.ok$

Fig. 1. Examples of histories: A history H, the subhistories $H|p$ and $H|T_2$ of H, and a history H', which is equivalent to H

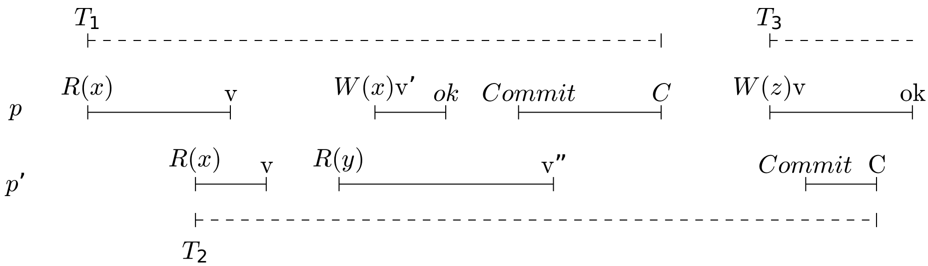

Fig. 2. A schematic representation of H presented in Figure 1. The horizontal axis represents time.

it is *live*. A transaction T is *commit-pending* in H if T is live in H and $H|T$ includes an invocation to COMMIT for T. If $H|T$ contains at least one invocation of WRITE, T is called an *update* transaction; otherwise, T is *read-only*. We denote by $comm(H)$ the subsequence of all events in H issued and received for committed transactions.

For each thread p, we denote by $H|p$ the subsequence of H containing all invocations and responses of t-operations issued or received by p. Two histories H and H' are *equivalent*, if for each thread p, $H|p = H'|p$. Roughly speaking, two histories H and H' are equivalent if they contain the same set of transactions, and each t-operation invoked in H is also invoked in H' and receives the same response in both H and H'. This means that the order of invocation and response events may be different in H' compared to H, although the orders of invocation and response events are the same in $H|p$ and $H'|p$ for each thread p. An example of history equivalence is presented in Figure 1. It shows H as a sequence of invocation and response events, and presents H', which is a history equivalent to H. History H is further illustrated in Figure 2.

We denote by $Complete(H)$ a set of histories that extend H. Specifically, a history H' is in $Complete(H)$ if and only if, all of the following hold:

1. H' is well-formed, H is a prefix of H', and H and H' contain the same set of transactions;
2. for every live transaction[1] T in H:
   (a) if $H|T$ ends with an invocation of COMMIT, H' contains either C_T or A_T;
   (b) if $H|T$ ends with an invocation other than COMMIT, H' contains A_T;
   (c) if $H|T$ ends with a response, H' contains ABORT_T and A_T.

Roughly speaking, each history in $Complete(H)$ is an extension of H where some of the commit-pending transactions in H appear as committed and all other live transactions appear as aborted. We say that H is *complete* if all transactions in H are complete. Each history in $Complete(H)$ is complete.

Given an execution α, the history of α, denoted by H_α, is the subsequence of α consisting of just the invocations and the responses of t-operations. The *execution interval* of a complete transaction T in an execution α is the subsequence of consecutive steps of α starting with the first step executed by any of the t-operations invoked by T and ending with the last such step. The *execution interval* of a transaction T that does not complete in α is the suffix of α starting with the first step executed by any of the t-operations invoked by T. We remark that similar definitions to the ones given on the base of histories, can also be given for executions: We say that a t-operation is complete in some execution α if it is complete in H_α; otherwise it is pending. A transaction T is committed (res. live, commit-pending) in α if it is committed (res. live, commit-pending) in H_α, etc.

Real-Time Order on Transactions and Sequential Histories. Consider a well-formed history H. We define a partial order, called *real time order* and denoted $<_H$, on the set of *transactions* in H, as follows:

- for any two transactions T_1 and T_2 in H, if T_1 is complete in H and the last event of $H|T_1$ precedes the first event of $H|T_2$ in H, then $T_1 <_H T_2$.

Transactions T_1 and T_2 are *concurrent* in H, if neither $T_1 <_H T_2$ nor $T_2 <_H T_1$. Similarly, transactions T_1 and T_2 are *concurrent* in an execution α, if neither $T_1 <_{H_\alpha} T_2$ nor $T_2 <_{H_\alpha} T_1$. We say that a history H (or an execution α) is *sequential* if no two transactions in H (in α) are concurrent.

Legality. Consider a sequential history S and a transaction T in S. We say that T is *legal* in S, if for every invocation *inv* of READ on each data item x that T performs, whose response is $res \neq A_T$, the following hold:

1. if there is an invocation of WRITE for x by T that precedes *inv* in S then *res* is the value argument of the last such invocation,
2. otherwise, if there are no committed transactions preceding T in S which invoke WRITE for x, then *res* is the initial value for x,

[1] We remark that the order in which the live transactions of H are inspected to form H' is immaterial, i.e. all histories that result by processing the live transactions in any possible such order are added to $Complete(H)$.

3. otherwise, *res* is the value argument of the last invocation of WRITE with parameter x, by any committed transaction that precedes T in S.

A complete sequential history S is *legal* if every transaction in S is legal.

Real-Time Order on t-operations and Operation-wise Sequential Histories. We define a partial order, called *operation real-time* order and denoted by $<_H^{op}$, on the set of *t-operations* in H, as follows:

- for any two t-operations op_1 and op_2 in H, if H contains a response for op_1 which precedes the invocation of op_2, then $op_1 <_H^{op} op_2$.

Operations op_1 and op_2 are *concurrent* in H, if neither $op_1 <_H^{op} op_2$ nor $op_2 <_H^{op} op_1$. H is *operation-wise sequential* if no two t-operations in H are concurrent.

Let S_{op} be an operation-wise sequential history equivalent to H. Since S_{op} is equivalent to H, S_{op} and H contain the same set of transactions. We say that S_{op} *respects* some relation $<$ on the set of *transactions* in H if the following holds: for any two transactions T_1 and T_2 in S_{op}, if $T_1 < T_2$, then $T_1 <_{S_{op}} T_2$. We say that S_{op} respects some relation $<^{op}$ on the set of *t-operations* in H if the following holds: for any two t-operations op_1 and op_2 in S_{op}, if $op_1 <^{op} op_2$, then $op_1 <_{S_{op}}^{op} op_2$. Notice that a partial order is a relation, so these definitions hold for partial orders as well.

3 TM Consistency

In this section, we present a collection of consistency conditions for STM computing.

3.1 Strict Serializability

Strict serializability was first introduced in [29] as a (strong) consistency condition for executions of concurrent transactions in database systems. Roughly speaking, an execution α is strictly serializable if each complete transaction that does not abort (as well as some of the live transactions) is executed in α like if it was executed serially at some point within its execution interval. A special case of strict serializability where transactions are restricted to consist of a single t-operation applied to a single data item is known as linearizability [22].

In STM computing, strict serializability can be expressed in several different flavors, two of which are discussed below. We start with *eager strict serializability* (or *e-strict serializability* for short).

Definition 1 (e-Strict Serializability). *We say that an execution α is* e-strictly serializable *if it is possible to do all of the following:*

- *If A is the set of all complete transactions in α that are not aborted, for each transaction $T \in A$, to associate with T a point $*_T$ somewhere between T's first invocation of a t-operation and T's last response of a t-operation in α.*
- *To choose a subset B of the* **live** *transactions in α and, for each transaction $T \in B$, associate with T a point $*_T$ somewhere after T's first invocation of a t-operation in α.*

For each $T \in A \cup B$, $*_T$ is called the serialization point *of T. Let σ be the sequential execution we get by serially executing (the code of) each transaction $T \in A \cup B$ at the place that its serialization point has been selected in α starting from the initial configuration. The set B and the serialization points of transactions in $A \cup B$ should be selected so that:*

- *for each transaction $T \in A$, the same t-operations, as in α, are invoked by T in σ and the response of each such t-operation in σ is the same as that in α, and*
- *for each transaction $T \in B$, a prefix of the t-operations invoked by T in σ is the same as the sequence of t-operations invoked by T in α and the response of each such t-operation in σ is the same as that in α (if it exists in α).*

An STM algorithm is e-strictly serializable *if each execution it produces is e-strictly serializable.*

If an execution α is e-strictly serializable, there exists a sequential execution σ (and a set B of live transactions in α) that satisfies the properties of Definition 1; we say that σ (and B) *justifies* that α is e-strictly serializable. Notice that since σ is the sequential execution produced by serially executing (the code of) each transaction at its serialization point starting from an initial configuration, σ is a legal execution and each transaction $T \in B$ commits in σ. Moreover, H_σ is a legal history containing only committed transactions.

We continue to provide a stronger version of e-strict serializability in Definition 2, called *deferred-update strict serializability* (or *du-strict serializability* for short), which is based on the definition of $Complete$.

Definition 2 (du-Strict Serializability, expressed in terms of histories). *A history H is* du-strictly serializable*, if there exist a history $H' \in Complete(H)$ and a history S equivalent to $comm(H')$ such that:*

- *S is a legal sequential history, and*
- *S respects $<_{comm(H')}$.*

An execution α is du-strictly serializable, if H_α is du-strictly serializable. An STM algorithm is du-strictly serializable*, if each execution α it produces is du-strictly serializable.*

Definition 2 follows the standard technique, employed in STM theory research, of presenting consistency conditions in terms of histories. We remark that this is not straightforward to achieve when defining the e-version of a consistency condition since in the e-version, serialization points can be associated even with live transactions (that are not commit-pending) for which it is unknown which t-operations they would invoke if they were to continue their execution until they complete. For compatibility with Definition 1, we present below, in Definition 3, du-strict serializability in terms of executions.

Definition 3 (du-Strict Serializability, expressed in terms of executions). *We say that an execution α is* du-strictly serializable *if it is possible to do all of the following:*

- *If A is the set of all complete transactions in α that are not aborted, for each transaction $T \in A$, to associate with T a point $*_T$ somewhere between T's first invocation of a t-operation and T's last response of a t-operation in α.*
- *To choose a subset B of the* **commit-pending** *transactions in H_α and, for each transaction $T \in B$, associate with T a point $*_T$ somewhere after T's first invocation of a t-operation in α.*

For each $T \in A \cup B$, $*_T$ is called the serialization point *of T. Let σ be the sequential execution we get by serially executing (the code of) each transaction $T \in A \cup B$ at the place that its serialization point has been selected in α starting from the initial configuration. The set B and the serialization points of transactions in $A \cup B$ should be selected so that:*

- *for each transaction $T \in A \cup B$, the same t-operations, as in α, are invoked by T in σ and the response of each such t-operation (other than* COMMIT*) in σ is the same as that in α.*

An STM algorithm is du-strictly serializable *if each execution it produces is du-strictly serializable.*

Lemma 1 argues that Definitions 2 and 3 are equivalent. Its proof is heavily based on the definitions of the concepts employed in Definitions 2 and 3.

Lemma 1. *Definitions 2 and 3 are equivalent in whatever concerns du-strictly serializable executions and STM algorithms.*

Sketch of proof. For the purpose of the proof, we will call an execution (or history) which satisfies the properties of Definition 2, *history-based du-ss*. Similarly, we will call an execution (or a history) which satisfies the properties of Definition 3, *execution-based du-ss*.

1. Consider an execution α which is history-based du-ss. We prove that α is execution-based du-ss.

   Since α is history-based du-ss, Definition 2 implies that H_α is history-based du-ss. Specifically, there exists a history $H' \in Complete(H_\alpha)$ and a history S equivalent to $comm(H')$ such that:

   - S is a legal sequential history, and
   - S respects $<_{comm(H')}$.

   By definition of $Complete(H_\alpha)$, H' is an extension of H_α where some of the commit-pending transactions in H_α appear as committed and all other live transactions appear as aborted. Let B be those commit-pending transactions in H_α that are committed in H', and let A be the set of all complete transactions in α (which are the same as in H_α) that do not abort. By definition of *comm*, $comm(H')$ is the subsequence of all events in H' issued and received for committed transactions, i.e. $comm(H')$ is the subsequence of all events issued or received for transactions in $A \cup B$.

   Since S is equivalent to $comm(H')$, S contains all transactions in $A \cup B$ (and no more transactions), and thus all transactions in S commit. Since S is sequential, it defines a total order on all transactions in $comm(H')$. Since S is equivalent to $comm(H')$ and respects $<_{comm(H')}$, it is possible to do the following: (1) for each transaction $T \in A$, to assign a serialization point for T somewhere between T's first invocation of a t-operation and T's last response of a t-operation in α, and (2) for each transaction $T \in B$, to assign a serialization point for T somewhere after T's first invocation of a t-operation in α, *so that* the total order defined by the serialization points on transactions in $A \cup B$ to be the same as that defined on transactions by S.

Let σ be the sequential execution, starting from the initial configuration, in which each transaction in S is serially executed, in the order it appears in S. Since S is legal, it is a straightforward induction to prove that, each transaction invokes the same t-operations in σ as in S and for each such invocation *inv*, *inv* has the same response in σ as in S. Thus, σ justifies that α is execution-based du-ss.

2. Now consider an execution β which is execution-based du-ss. We prove that β is history-based du-ss.

   Let A be the set of complete transactions in β that are not aborted, and let B and σ be the set of commit-pending transactions in β and the sequential execution, respectively, that justify the (execution-based) du-ss property of β. Let H' be an extension of H_β which is constructed as follows: (1) for each commit-pending transaction $T \in B$ we add a C_T response, and (2) for each other live transaction T in β we add an A_T response. Then, $H' \in Complete(H_\beta)$ and $comm(H')$ is the subsequence of H' containing all events issued or received for transactions in $A \cup B$.

   Let $S = H_\sigma$. Since σ is the sequential execution produced by serially executing (the code of) each transaction in $A \cup B$ at its serialization point, σ is a legal execution and each transaction $T \in A \cup B$ commits in σ. Thus, S is a legal sequential history which contains all transactions in $A \cup B$ (and no further transactions), and all these transactions commit in S. Since for each transaction $T \in A \cup B$, the same t-operations, as in β (or in H_β), are invoked by T in σ (or in H_σ) and the response of each such t-operation (other than COMMIT) in H_σ is the same as that in H_β, it follows that S is equivalent to $comm(H')$.

   Since (1) for each transaction $T \in A$, $*_T$ is placed between T's first invocation of a t-operation and T's last response of a t-operation in β, and (2) for each transaction $T \in B$, $*_T$ is placed somewhere after T's first invocation of a t-operation in β, it follows that $S = H_\sigma$ respects $<_{comm(H')}$. So, H' and S justify that H_β is history-based du-ss. Therefore, β is history-based du-ss.

□

Since a commit-pending transaction is live, it is straightforward to see that Definition 1 provides a weaker version of strict serializability than Definition 3 (or Definition 2). Intuitively, this is so since Definition 1 allows a transaction to read a value for a data item written by another transaction that is not committed or commit-pending in H. (This is allowed only if eventually, all complete transactions that are not aborted, and some of those that are still live can be "serialized" within their execution intervals.) It follows that if an execution is du-strictly serializable, it is also e-strictly serializable. However, the opposite is not true. For instance, let's consider the history H and its prefix

H_1 both shown in Figure 3. H is both e-strictly serializable and du-strictly serializable, whereas H_1 is just e-strictly serializable.

Lemma 2. *If an execution α is du-strictly serializable then α is e-strictly serializable, but not vice versa.*

A set $\mathscr{S}$ of sequences is prefix-closed if, whenever H is in $\mathscr{S}$, every prefix of H is also in $\mathscr{S}$. Recall that the history H shown in Figure 3 is du-strictly serializable but its prefix H_1 is not. Thus, du-serializability is not a prefix-closed property. On the contrary, e-strict serializability is a prefix-closed property. We remark that prefix-closure can be imposed to du-strict serializability in an explicit way, i.e. by directly stating in Definition 2 that each prefix H_p of H must also satisfy the conditions imposed by the definition. This would make Definition 2 stronger.

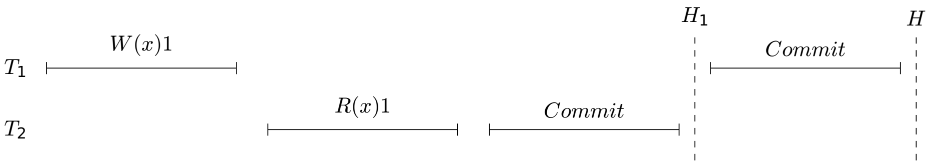

Fig. 3. Example of a history H showing that du-strict serializability is not a prefix-closed property. We remark that H is operation-wise sequential. In all our example histories, we assume that the initial value of each of the employed data items is 0.

3.2 Serializability

As with strict serializability, serializability was first introduced in [29] as a consistency condition for executions of concurrent transactions in database systems. It is weaker than strict serializability in that it does not ensure that the serialization point of each transaction is within its execution interval. Below, we discuss two different flavors of serializability in a way similar to that for strict serializability.

Definition 4 (e-Serializability). *We say that an execution α is* e-serializable *if it is possible to do all of the following:*

- *If A is the set of all complete transactions in α that are not aborted, for each transaction $T \in A$, to associate with T a point $*_T$ in α.*
- *To choose a subset B of the live transactions in α and, for each transaction $T \in B$, to associate with T a point $*_T$ in α.*

For each $T \in A \cup B$, $*_T$ is called the serialization point *of T. Let σ be the sequential execution we get by serially executing (the code of) each transaction $T \in A \cup B$ at the place that its serialization point has been selected in α starting from the initial configuration. The set B and the serialization points of transactions in $A \cup B$ should be selected so that:*

- *for each transaction $T \in A$, the same t-operations, as in α, are invoked by T in σ and the response of each such t-operation in σ is the same as that in α, and*

- *for each transaction $T \in B$, a prefix of the t-operations invoked by T in σ is the same as the sequence of t-operations invoked by T in α and the response of each such t-operation in σ is the same as that in α (if it exists in α).*

An STM algorithm is e-serializable *if each execution it produces is e-serializable.*

We continue to provide a stronger version of serializability in Definition 5, called *deferred-update serializability* (or *du-serializability* for short), which is based on the definition of $Complete$.

Definition 5 (du-Serializability). *A history H is* du-serializable*, if there exists a history $H' \in Complete(H)$ and a history S equivalent to $comm(H')$ such that:*

- *S is a legal sequential history.*

An execution α is du-serializable, if H_α is du-serializable. An STM algorithm is du-serializable *if each execution α it produces is du-serializable.*

Notice that S in Definition 5 respects the program order of t-operations executed by the same thread in H. This is implied by the definition of equivalent histories.

We remark that, similarly to the corresponding definitions of strict serializability, Definition 4 provides a weaker version of serializability than Definition 5. This can be easily seen by deriving an execution-based version of Definition 5 (in the spirit of Definition 3) and proving that this version is equivalent to Definition 5 (as proved in Lemma 1 for du-strict serializability).

Lemma 3. *If an execution α is du-serializable then α is e-serializable, but not vice versa.*

The difference between serializability and strict serializability is that strict serializability additionally ensures that the real-time order of transactions is respected by the sequential history defined by the serialization points. Thus, every history/execution that is (du-) e-strictly serializable is also (du-) e-serializable but not vice versa.

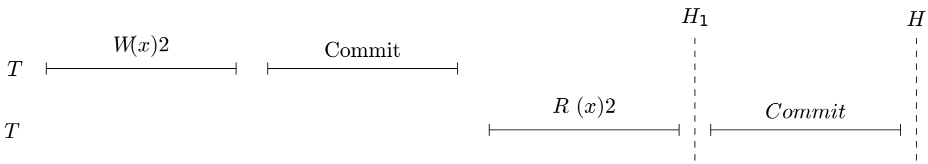

Fig. 4. Example showing that du-serializability is not a prefix-closed property

Lemma 4. *If an execution α is strictly serializable then α is serializable, but not vice versa*[2].

[2] When we say that an execution (or an STM algorithm) satisfies a consistency condition without specifying which variant of the condition it satisfies, then the claim holds for both variants of that condition.

It is worth pointing out that e-serializability and du-serializability are not prefix-closed properties. This is so, since it is easy to design a history H which is e-serializable (as well as du-serializable) in which a committed transaction T (executed by some thread p) reads for some data item x a value v and then commits. H also contains a second transaction T' (executed by some thread $p' \neq p$) which starts its execution after T has completed, writes v into x, and commits. Such a history is shown in Figure 4. We remark that H is e-serializable and du-serializable. However, the prefix of H up until C_T is neither e-serializable, nor du-serializable.

We remark that prefix-closure can be imposed to e-serializability (as well as to du-serializability) in an explicit way, as discussed for du-strict serializability above. It is not clear if the versions that would then result will be weaker than the corresponding versions of strict serializability. Imposing prefix closure to the consistency conditions presented in Sections 3.4-3.5 may be too restrictive as well. Thus, we present the non-prefix-closed versions of them given that it is straightforward to derive their prefix-closed versions, in an explicit way.

Several impossibility results [6,11,16] and lower bounds [6] in STM computing have been proved for strict serializability or serializability. Most STM algorithms in the literature (see e.g. [12,39,14,33] for some examples) satisfy some form of serializability.

3.3 Opacity

Opacity was first introduced in [18]. Definition 6 follows that in [18]. Roughly speaking, a history H that is du-opaque is also du-strictly serializable; additionally, if S is the sequential history which justifies that S is du-strictly serializable, opacity ensures that those transactions in H that are not included in S are also legal. For instance, such transactions are those that have aborted in H (but there may be more).

Definition 6 (du-Opacity [19]). *A history H is du-opaque if there exists a sequential history S equivalent to some history $H' \in Complete(H)$ such that:*

- *S is legal, and*
- *S respects $<_{H'}$.*

An execution α is du-opaque, if H_α is du-opaque. An STM algorithm is du-opaque *if each execution α it produces is du-opaque.*

In [19], a prefix-closed version of opacity was formally stated. According to it, a history H is du-opaque if the conditions imposed by Definition 6 are satisfied for each prefix H_p of H; this version of du-opacity is stronger than that provided in Definition 6 which is not prefix-closed. Figure 5 illustrates a situation that would be acceptable by the non-prefix-closed version of du-opacity. History H', which is a prefix of history H, does not satisfy du-opacity, as transaction T_2 reads a value written by a transaction which is still not committed. However, as transaction T_1 is committed in history H, H complies with du-opacity. A different formalization of du-opacity as a prefix-closed property was elaborated in [5].

Lemma 5 argues that du-opacity is stronger than du-strict serializability.

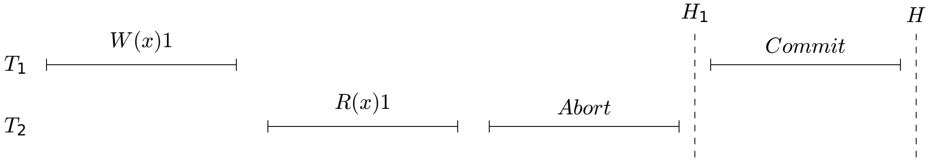

Fig. 5. Example showing that du-opacity is not a prefix-closed property

Lemma 5. *If an execution α is du-opaque, then α is du-strictly serializable, but not vice versa.*

Proof. Since α is du-opaque, H_α is also du-opaque. Thus, there exists a sequential history S, equivalent to some history $H' \in Complete(H_\alpha)$, such that S is legal and S respects $<_{H'}$.

Let S' be the subsequence of S consisting of all events in S issued or received by transactions in $comm(H')$. Then, the following hold:

- Since S is equivalent to H', it follows that S' is equivalent to $comm(H')$.
- Since S respects $<_{H'}$, it follows that S' respects $<_{comm(H')}$.
- Since S is legal, it follows that each transaction in S is legal. Since S is equivalent to H' and S' is comprised of the events of all transactions in $comm(H')$, it follows that S' is legal.

Thus, H' and S' satisfy the properties of Definition 2 and, therefore, H_α is du-strictly serializable. So, α is du-strictly serializable.

Figure 6 shows an example of a history that is not du-opaque but is du-strictly serializable. This history is not du-opaque because it violates the first condition of Definition 6; specifically, transaction T_2 cannot be legal.

Fig. 6. A du-strictly serializable history which is not du-opaque

Lemmas 2 and 5 imply the following corollary.

Corollary 1. *If an execution α is du-opaque, then α is e-strictly serializable.*

Consider an execution α which is du-strictly serializable, and let S be the sequential history that justifies that α is du-strictly serializable. Strict serializability doesn't impose any restrictions on those transactions in α that are not included in S, whereas (roughly speaking) du-opacity requires that all reads of each such transaction T (independently of whether the transaction is aborted or live in α) read values written by previously committed transactions (or by T itself). This additional property is required in order to avoid undesired situations where a transaction may cause an exception or enter into an infinite loop after reading a value for a data item written by a live transaction that may eventually abort.

It is remarkable that the first of these undesired situations (i.e. the production of an exception or an error code) can be avoided even by STM systems that ensure only strict serializability if we make the following simple assumptions in our model. An exception (or an error code) that has been resulted by the execution of a t-operation op is considered as a response for op. A transaction that has experienced an exception or has received an error code as a response to one of its t-operations, is considered to be complete (but not aborted). Then, a (e- or du-) strictly serializable STM implementation will never produce such exceptions (or error codes). Notice that the second undesirable situation, namely having some transaction enter an infinite loop, will not appear in STM systems that ensure standard progress properties, like lock-freedom, starvation-freedom, etc. A thread p experiences *starvation* in an execution α, if p takes infinitely many steps in α and it receives only a finite number of commit responses for the transactions that it initiates; an STM algorithm is *starvation-free*, if, in every execution that it produces, no thread ever experiences starvation. *Obstruction-freedom* ensures that for each thread p, if p runs solo starting from any configuration C in α, it eventually completes the execution of its transaction successfully within a finite number of steps.

We continue to present eager opacity (e-opacity). Consider any history H and a transaction T in H. An instance op of READ for some data item x executed by T is *global* if T has not invoked WRITE on x in H before invoking op. Let $H|T|read$ be the longest subsequence of $H|T$ consisting of those invocations of READ (and their responses) for which there is a response and this response is not A_T, followed by COMMIT_T, C_T. Let $H|T|read_g$ be the subsequence of $H|T|read$ consisting only of the invocations of the *global* instances of READ and their responses, followed by COMMIT_T, C_T. We denote by $T_r(H)$ a transaction that invokes the same t-operations (and in the same order) as those invoked in $H|T|read$. Similarly, denote by $T_{gr}(H)$ a transaction that invokes the same t-operations (and in the same order) as those invoked in $H|T|read_g$. $T_r(H)$ and $T_{gr}(H)$ are defined for an execution α in terms of H_α. For each READ t-operation op on any data item x that is in $T_r(H)$ ($T_r(\alpha)$) but not in $T_{gr}(H)$ ($T_{gr}(\alpha)$), we say that the response for op (if it exists) is *legal*, if it is the value written by the last WRITE for x performed by T before the invocation of op.

Definition 7 (e-Opacity). *We say that an execution α is* e-opaque *if there exists a set B of live transactions in α and some sequential execution σ which justify that α is e-strictly serializable, and all of the following hold:*

1. *We can extend the history H_σ of σ to get a* sequential *history H'_σ such that:*
   - *if A is the set of complete transactions in α that are not aborted, for each transaction T in α that is not in $A \cup B$ (i.e. for each transaction T in α that is not in σ), H'_σ contains $H_\alpha|T|read_g$,*
   - *if $<$ is the partial order which is induced by the real time order $<_{H_\alpha}$ in such a way that for each transaction T in α that is not in σ, we replace each instance of T in the set of pairs of $<_{H_\alpha}$ with transaction $T_{gr}(\alpha)$, then H'_σ respects $<$, and*

- *H'_σ is legal.*

2. *For each transaction T in α that is not in σ, and for each invocation of a* READ *operation op which is in $H_\alpha|T|read$ but not in $H_\alpha|T|read_g$, the response for op is legal.*

An STM algorithm is e-opaque *if each execution α it produces is e-opaque.*

Lemma 6 proves that du-opacity is stronger than e-opacity.

Lemma 6. *If an execution α is du-opaque, then α is e-opaque, but not vice versa.*

Sketch of proof. Since α is du-opaque, H_α is also du-opaque. Thus, there exists a sequential history S, equivalent to some history $H' \in Complete(H_\alpha)$, such that S is legal and S respects $<_{H'}$.

Let S' be the subsequence of S consisting of all events in S issued or received by transactions in $comm(H')$. Then, by following similar arguments as in the proof of Lemma 5 we argue that H' and S' satisfy the properties of Definition 2 and, therefore, H_α is du-strictly serializable. So, α is du-strictly serializable.

Let B be those commit-pending transactions in H_α that are committed in H'. Let σ be the sequential execution, starting from the initial configuration, in which each transaction in S' is serially executed, in the order it appears in S'. We follow similar arguments as in the proof of Lemma 1 to argue that σ justifies that α is execution-based du-ss. Thus, Lemma 2 implies that α is e-strictly serializable; moreover, we argue that $H_\sigma = S'$.

Denote by A the set of complete transactions in H_α that are not aborted. Let H'_σ be the subsequence of S such that H'_σ contains all events in S', and for each transaction $T \notin A \cup B$, H'_σ additionally contains each event in $H|T|read_g$. Since S and S' are legal, it follows that H'_σ is also legal. Also, since S respects $<_{H'}$, it follows that S' respects $<$ (as defined in item 1 of Definition 7). Thus, H'_σ (which is equal to S') respects $<$.

Finally, legality of S implies that for each transaction T in α that is not in σ, and for each t-operation op in $T|read$ that is not in $T|read_g$, the response for op is legal. We conclude that α is e-opaque.

Figure 6 shows an example of a history that is not du-opaque but is du-strictly serializable (and therefore also e-strictly serializable, by Lemma 2). This history is not du-opaque because it violates the first condition of Definition 6; specifically, transaction T_2 cannot be legal. □

We remark that most STM algorithms presented in the literature are opaque.

3.4 Causality-Related Consistency Conditions

Consider any operation-wise sequential history S_{op} that is equivalent to H. Since S_{op} is equivalent to H, there are the same transactions in S_{op} as in H. We define a binary relation with respect to S_{op}, called *reads-from* and denoted by $<^r_{S_{op}}$, between *transactions* in H such that, for any two transactions T_1, T_2 in H, $T_1 <^r_{S_{op}} T_2$ only if:

- T_2 executes a READ t-operation op that reads some data item x and returns a value v for it,

- T_1 is the transaction in S_{op} which executes the last WRITE t-operation that writes v for x and precedes op.

Notice that each operation-wise sequential history S_{op} that is equivalent to H, induces a *reads-from* relation for H. We denote by $\mathscr{R}_H$ the set of all reads-from relations that can be induced for H.

For each $<^r$ in $\mathscr{R}_H$, we define the *causal* relation for $<^r$ on transactions in H to be the transitive closure of $\bigcup_i \left(<_{H|p_i}\right) \cup <^r$. We define $\mathscr{C}_H$ to be the set of all causal relations in H.

Causal Consistency. Causal consistency was informally introduced as a shared memory consistency condition in [23], and it was formally defined in [2]. Roughly speaking, an execution α is causally consistent if for each thread p_i, there exists a sequential execution σ_i of the complete transactions that are not aborted (as well as of some of the live transactions) in α such that in σ_i each of these transactions invokes the same t-operations and gets the same responses as in α. Thus, causal consistency allows the sequential executions to be different for different threads. However, it imposes the additional constraint that all sequential executions respect the same causal relation.

As in the previous sections, we provide two formal definitions of causal consistency for STM computing.

Definition 8 (e-Causal Consistency). *Consider an execution α and let A be the set of all complete transactions in α that are not aborted. We say that α is* e-causally-consistent *if there exists a subset B of live transactions in α and a causal relation $<^c$ in $\mathscr{C}_{H'_\alpha}$, where H'_α is the subsequence of H_α consisting of the events (in H_α) issued and received for the transactions in $A \cup B$, such that,* for each thread p_i, *it is possible to do the following:*
For each transaction $T \in A \cup B$, to associate with T a point $*^i_T$ in α. Let σ_i be the sequential execution we get by serially executing (the code of) each transaction $T \in A \cup B$ at the place that its point has been selected (for p_i) in α starting from the initial configuration. The set B, and the points of transactions in $A \cup B$ should be selected (for p_i) so that:

- *H_{σ_i} respects $<^c$,*
- *for each transaction $T \in A$, the same t-operations, as in α, are invoked by T in σ_i and the response of each such t-operation in σ_i is the same as that in α, and*
- *for each transaction $T \in B$, a prefix of the t-operations invoked by T in σ_i is the same as the sequence of t-operations invoked by T in α, and the response of each such t-operation in σ_i is the same as that in α (if it exists in α).*

An STM algorithm is e-causally-consistent *if each execution α it produces is e-causally-consistent.*

We continue with the presentation of the du-version of causal consistency.

Definition 9 (du-Causal Consistency). *A history H is* du-causally consistent *if there exists a history $H' \in Complete(H)$ and a causal relation $<^c$ in $\mathscr{C}_{comm(H')}$ such that, for each thread p_i, there exists a sequential history S_i such that:*

- *S_i is equivalent to $comm(H')$,*
- *S_i respects the causality order $<^c$, and*
- *every transaction executed by p_i in S_i is legal.*

An execution α is du-causally consistent, *if H_α is du-causally consistent. An STM algorithm is* du-causally consistent *if each execution α it produces is du-causally consistent.*

By following similar arguments as in the proof of Lemma 2, it can be proved that du-causal consistency is stronger than e-causal consistency.

Lemma 7. *If an execution α is du-causally consistent then α is e-causally consistent, but not vice versa.*

Lemma 8 argues that serializability is stronger than causal consistency.

Lemma 8. *If an execution α is serializable then α is causally consistent, but not vice versa.*

Sketch of proof. We prove the claim for the e-versions of the consistency conditions. The proof of the claim for the du-variants of them can be performed using similar reasoning.

Let A be the set of complete transactions in α that are not aborted. Moreover, let B and σ be the set of live transactions in α and the sequential execution, respectively, which justify that α is serializable. By Definition 4, the following hold for σ:

- for each transaction $T \in A$, the same t-operations, as in α, are invoked by T in σ and the response of each such t-operation in σ is the same as that in α, and
- for each transaction $T \in B$, a prefix of the t-operations invoked by T in σ is the same as the sequence of t-operations invoked by T in α, and the response of each such t-operation in σ is the same as that in α (if it exists in α).

Let H'_σ be the subsequence of H_σ in which, for each transaction $T \in B$, we exclude those events issued or produced for T in σ that are not in α. Then, H'_σ is equivalent to H'_α, where H'_α is the subsequence of H_α containing just the events of transactions in $A \cup B$. Since H'_σ is sequential, it is also operation-wise sequential, so $<^r_{H'_\sigma}$ is well-defined. Let $<^c$ be the causal relation for $<^r_{H'_\sigma}$. Then, by letting $\sigma_i = \sigma$, for each thread p_i, all conditions of Definition 8 hold.

Figure 7 shows an example of a history which is du-causally consistent (and therefore also e-causally consistent, by Lemma 7) but not e-serializable. In this history both transactions T_1 and T_2 should be serialized before transactions T_3 and T_4, because both T_1 and T_2 read 0 from data item y which is written by T_3 and T_4. Regardless of how the serialization points for T_1 and T_2 are ordered, both T_3 and T_4 should read the same value for data item x. Thus, this history is not e-serializable (and therefore it is not e-serializable, by Lemma 3). However, it is du-causally consistent because threads running T_3 and T_4 may see writes executed by threads running T_1 and T_2 in a different order.

$\square$

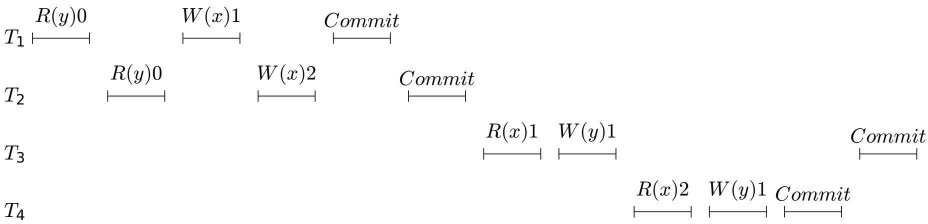

Fig. 7. A du-causally consistent history which is not e-serializable

Causal Serializability. Causal serializability was introduced in [31] as a consistency condition which is stronger than causal consistency but weaker than serializability. Informally, in addition to the constraints imposed by causal consistency, the following constraint must also be satisfied: all transactions that update the same data item must be perceived in the same order by all threads.

Definition 10 (e-Causal Serializability). *Consider an execution α and let A be the set of all complete transactions in α that are not aborted. We say that α is* e-causally serializable *if there exists a subset B of live transactions in α and a causal relation $<^c$ in $\mathscr{C}_{H'_\alpha}$ where H'_α is the subsequence of H_α consisting of the events (in H_α) issued and received for the transactions in $A \cup B$, such that,* for each thread p_i, *it is possible to do the following:*
For each transaction $T \in A \cup B$, to associate with T a point $*^i_T$ in α. Let σ_i be the sequential execution we get by serially executing (the code of) each transaction $T \in A \cup B$ at the place that its point has been selected (for p_i) in α starting from the initial configuration. The set B, and the points of transactions in $A \cup B$ should be selected (for p_i) so that:

- *H_{σ_i} respects $<^c$,*
- *for each transaction $T \in A$, the same t-operations, as in α, are invoked by T in σ_i and the response of each such t-operation in σ_i is the same as that in α,*
- *for each transaction $T \in B$, a prefix of the t-operations invoked by T in σ_i are the same as the sequence of t-operations invoked by T in α, the response of each such t-operation in σ_i is the same as that in α (if it exists in α).*
- *for each pair of transactions $T_1, T_2 \in A \cup B$ that write to the same data item, if $T_1 <_{H_{\sigma_i}} T_2$, then for each $j \in \{1, \ldots, n\}$, it holds that $T_1 <_{H_{\sigma_j}} T_2$.*

An STM algorithm is e-causally serializable *if each execution α it produces is e-causally serializable.*

We continue with the presentation of the du-version of causal serializability.

Definition 11 (du-Causal Serializability). *A history H is* du-causally serializable *if there exists a history $H' \in Complete(H)$ and a causal relation $<^c$ in $\mathscr{C}_{comm(H')}$ such that, for each thread p_i, there exists a sequential history S_i for which the following hold:*

- *S_i is equivalent to $comm(H')$,*
- *S_i respects the causality order $<^c$,*
- *every transaction executed by p_i in S_i is legal, and*
- *for each pair of transactions T_1 and T_2 in $comm(H')$ that write to the same data item, if $T_1 <_{S_i} T_2$, then for each $j \in \{1,\ldots,n\}$, it holds that $T_1 <_{S_j} T_2$.*

An execution α is du-causally serializable*, if H_α is du-causally serializable. An STM algorithm is* du-causally serializable *if each execution α it produces is du-causally serializable.*

By following similar arguments as in the proof of Lemma 2, it can be proved that du-causal serializability is stronger than e-causal serializability.

Lemma 9. *If an execution α is du-causally serializable then α is e-causally serializable, but not vice versa.*

Obviously, every (e- or du-) causally serializable history satisfies the properties of (e- or du-, respectively) causal consistency, but the opposite is not true. For instance, the du-causally consistent history shown in Figure 7 is not e-causally serializable, since threads executing transactions T_3 and T_4 do not see writes from T_1 and T_2 to data item x in the same order.

Lemma 10. *If an execution α is causally serializable then α is causally consistent, but not vice versa.*

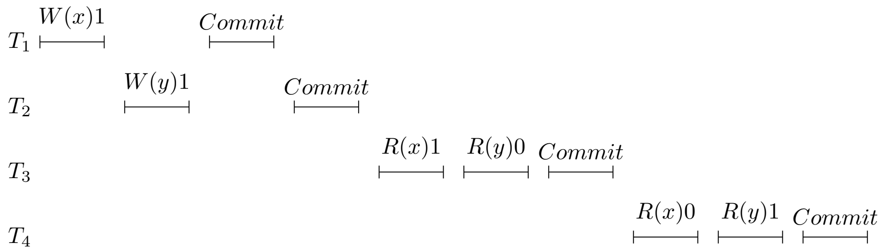

Fig. 8. A du-causally serializable history which is not e-serializable

Using similar arguments as those in the proof of Lemma 8, it can be easily proved that causal serializability is weaker than serializability. However, the opposite does not hold. Figure 8 shows an example of a history H which is du-causally serializable (and therefore also e-causally serializable, by Lemma 9) but not e-serializable (and therefore not du-serializable, by Lemma 3). In H, if transaction T_1 is serialized before T_2 (the opposite case is symmetrical), then it is not possible to serialize transaction T_4. However, by definition of causal serializability, sequential histories constructed for threads p_3 and p_4 may include transactions T_1 and T_2 in different orders.

Lemma 11. *If an execution α is serializable then α is causally serializable, but not vice versa.*

In STM research, causal consistency, as well as causal serializability, are interesting in the context of proving impossibility results [9,10] and lower bounds. We remark that when proving such results, considering a weak consistency condition makes the result stronger. It is therefore an interesting open problem to see whether some of the STM impossibility results (e.g. [6,11,16]) that have been proved assuming some strong consistency condition, like opacity, strict serializability or serializability, can be extended to hold for weaker consistency conditions like those formulated in this or later sections. For instance in this avenue, the impossibility result proved in [17] assuming serializability is extended in [9,10] to hold for a much weaker consistency condition.

Virtual World Consistency. Virtual World Consistency (VWC) was defined in [24] as a family of consistency conditions. Informally, VWC ensures serializability or strict serializability for the committed (and some of the commit-pending) transactions but a weaker condition than that imposed by opacity for the rest of the transactions.

For each transaction T in history H and each causal relation $<^c_H$ in $\mathscr{C}_H$, we define the *causal past* of T denoted by $past_T(H,<^c_H)$ as the subsequence of all events of H issued or produced either for transaction T itself or for any transaction T_i in H such that $T_i <^c_H T$.

Definition 12 (du-Virtual World Consistency). *A history H is* du-virtual world consistent *if there exists a history $H' \in Complete(H)$ and a causal relation $<^c$ in $\mathscr{C}_{H'}$ such that:*

- *there exists a legal sequential history S which is equivalent to $comm(H')$, and*
- *for each transaction T_i in H' that is not in S, there exists a legal sequential history S_i which is equivalent to $past_{T_i}(H',<^c)$ and respects the restriction of $<^c$ to those pairs whose components are transactions in $past_{T_i}(H',<^c)$.*

An execution α is du-virtual world consistent, *if H_α is du-virtual world consistent. An STM algorithm is* du-virtual world consistent *if each execution α it produces is du-virtual world consistent.*

Definition 13 (du-Strong Virtual World Consistency). *A history H is* du-strongly virtual world consistent *if there exists a history $H' \in Complete(H)$ and a causal relation $<^c$ in $\mathscr{C}_{H'}$ such that:*

- *there exists a legal sequential history S which is equivalent to $comm(H')$ and respects $<_{comm(H')}$, and*
- *for each transaction T_i in H' that is not in S, there exists a legal sequential history S_i which is equivalent to $past_{T_i}(H',<^c)$ and respects the restriction of $<^c$ to those pairs whose components are transactions in $past_{T_i}(H',<^c)$.*

An execution α is du-strongly virtual world consistent, *if H_α is du-strongly virtual world consistent. An STM algorithm is* du-strongly virtual world consistent *if each execution α it produces is du-strongly virtual world consistent.*

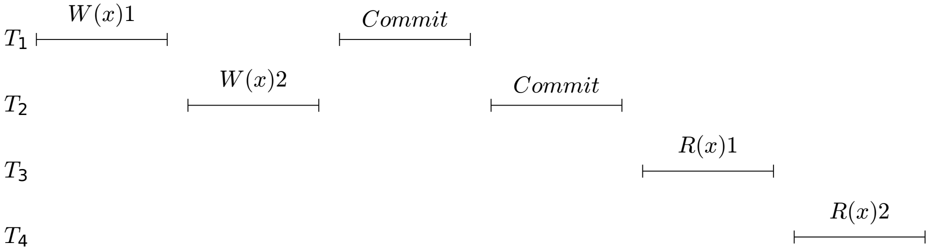

Fig. 9. A du-virtual world consistent history which is not du-opaque

By comparing Definitions 12 and 13 with Definitions 5 and 2, respectively, it is straightforward to see that du-virtual world consistency is stronger than du-serializability and du-strong virtual world consistency is stronger than du-strict serializability.

Lemma 12. *If an execution α is du-virtual world consistent (du-strongly virtual world consistent) then α is du-serializable (du-strictly serializable), but not vice versa.*

Du-strong virtual world consistency (and therefore also du-virtual world consistency) is weaker than du-opacity.

Lemma 13. *If an execution α is du-opaque then α is du-strongly virtual world consistent, but not vice versa.*

Sketch of proof. Since α is du-opaque, H_α is also du-opaque. Thus, there exists a sequential history S, equivalent to some history $H' \in Complete(H_\alpha)$, such that S is legal and S respects $<_{H'}$. Let S' be the subsequence of S consisting of all events in S issued or received by transactions in $comm(H')$. Then, S' is a legal sequential history, equivalent to $comm(H')$, which respects $<_{comm(H')}$.

Since S is sequential, it is also operation-wise sequential, so $<^r_S$ is well-defined. Let $<^c$ be the causal relation for $<^r_S$. Consider any transaction T_i in H' that is not in S'. Then, $past_{T_i}(H', <^c)$ is the subsequence of all events of H' issued or produced either for transaction T_i itself or for any transaction T_j in H' such that $T_j <^c T_i$.

Let S_i be the subsequence of S consisting of all events issued or produced for transactions in $past_{T_i}(H', <^c)$. Since S is equivalent to H', it follows that S_i is equivalent to $past_T(H', <^c)$. Since S_i is a subsequence of S and $<^c$ is the causal relation for $<^r_S$, it follows that S_i respects the restriction of $<^c$ to those pairs whose components are transactions in $past_{T_i}(H', <^c)$. Since S is legal and S_i is a subsequence of S equivalent to $past_T(H', <^c)$, it follows that S_i is legal. Thus, all conditions of Definition 13 hold.

The history shown in Figure 9 is du-strongly virtual world consistent but not du-opaque: regardless of the order of the serialization points of transactions T_1 and T_2, it is not possible to derive a sequential history where both transaction T_3 and T_4 are legal. □

We continue to present the eager versions of virtual world consistency and strong virtual world consistency.

Definition 14 (e-Virtual World Consistency and e-Strong Virtual World Consistency). *We say that an execution α is* e-virtual world consistent *(*e-strongly virtual world consistent*) if there exists some sequential execution σ which justifies that α is e-serializable (e-strictly serializable, respectively), and the following holds:*

1. *for each transaction T_i in α that is not in σ there exists a legal sequential history S_i which is equivalent to $past_{T_i}(H', <^c)$ and respects the restriction of $<^c$ to those pairs whose components are transactions in $past_{T_i}(H', <^c)$.*

An STM algorithm is e-virtual world consistent *(*e-strongly virtual world consistent*) if each execution α it produces is e-virtual world consistent (e-strongly virtual world consistent).*

Using similar arguments as in the proof of Lemma 6, we can prove that du-virtual world consistency is stronger than e-virtual world consistency.

Lemma 14. *If an execution α is du-virtual world consistent (du-strongly virtual world consistent) then α is e-virtual world consistent (e-strongly virtual world consistent), but not vice versa.*

The following lemma is an immediate consequence of Definition 14.

Lemma 15. *If an execution α is e-virtual world consistent (e-strongly virtual world consistent) then α is e-serializable (e-strictly serializable), but not vice versa.*

Using similar reasoning as that in the proof of Lemma 13, we can prove that e-opacity is stronger than e-strong virtual world consistency.

Lemma 16. *If an execution α is e-opaque then α is e-strongly virtual world consistent, but not vice versa.*

Strong consistency conditions such as opacity ensure the safe execution of non-committed transactions by imposing on them the same correctness demands as those that committed transactions are required to obey. This has been criticized in [24] to result in STM algorithms that produce histories in which live transactions are forced to abort in order to preserve the consistency of other transactions that are deemed to also abort. Virtual world consistency relaxes the correctness property used for non-committed transactions in order to avoid such scenarios in several cases, and by consequence, allow for more live transactions to commit than an STM algorithm that implements a stronger consistency condition would.

3.5 Snapshot Isolation

Snapshot isolation was originally introduced as a consistency condition in the database world [7,28]. Snapshot isolation is an appealing property for STM computing [3,13,32] since it provides the potential to increase throughput for workloads with long transactions [32]. The first formal definitions for STM snapshot isolation was given in [9,10].

Consider a history H and let T be a transaction that either commits or is commit-pending in H. Recall that we have already defined the sequences $H|T|read$, $H|T|read_g$, as well as transactions $T_r(H)$ and $T_{gr}(H)$ in Section 3.3. Let $H|T|other$ be the subsequence of $H|T$ that consists of all invocations performed by T (and their matching responses) in H other than those comprising $H|T|read_g$, followed by COMMIT$_T$,C_T. Let $T_o(H)$ be a transaction that invokes the same t-operations (and in the same order) as those invoked in $H|T|other$; for an execution α $T_o(\alpha)$ is defined in terms of H_α in the same way.

Definition 15 (du-Snapshot isolation [10]). *An execution α satisfies* du-snapshot isolation, *if there exists a set D consisting of all committed and some of the commit-pending transactions in α for which the following holds:*
For each transaction $T \in D$, it is possible to insert (in α) a point $*_{T,gr}$, called the global read point *of T, and a point $*_{T,w}$, called the* write point *for T, so that if δ_α is the sequence defined by these serialization points, the following hold:*

1. *$*_{T,gr}$ precedes $*_{T,w}$ in δ_α,*
2. *both $*_{T,gr}$ and $*_{T,w}$ are inserted within the execution interval of T,*
3. *if H_{δ_α} is the history we get by replacing each $*_{T,gr}$ with $H_\alpha|T|read_g$ and each $*_{T,w}$ with $H_\alpha|T|other$ in δ_α, then H_{δ_α} is legal.*

An STM algorithm satisfies du-snapshot isolation *if each execution α it produces satisfies du-snapshot isolation.*

We now present eager snapshot isolation. Consider a legal execution α and let $C(\alpha)$ be the set of all legal executions such that each execution $\alpha' \in C(\alpha)$ is an extension of α such that the same transactions are executed in α and α' and no transaction is live in α'.

Definition 16 (e-Snapshot Isolation). *Consider an execution α. We say that α satisfies* e-snapshot isolation, *if there exists an execution $\alpha' \in C(\alpha)$ for which the following holds: if A is the set of transactions that commit in α' then for each transaction $T \in A$, it is possible to insert a point $*_{T,gr}$, called* global read point *of T, and a point $*_{T,w}$, called* write point *of T, in α, so that:*

1. *$*_{T,gr}$ precedes $*_{T,w}$,*
2. *both $*_{T,gr}$ and $*_{T,w}$ are inserted somewhere between T's first invocation of a t-operation and T's last response of a t-operation in α', and*
3. *if σ is the sequential execution that we get when for each transaction $T \in A$, we serially execute transactions $T_{gr}(\alpha)$ and $T_o(\alpha)$ at the points that $*_{T,gr}$ and $*_{T,w}$, respectively, have been inserted, then for each transaction $T \in A$, the response of each t-operation invoked by $T_{gr}(\alpha)$ and $T_o(\alpha)$ in σ is the same as that of the corresponding t-operation in $H_\alpha|T|read_g$ and $H_\alpha|T|other$, respectively.*

An STM algorithm satisfies e-snapshot isolation *if each execution α it produces satisfies e-snapshot isolation.*

Lemma 17 argues that du-snapshot isolation is stronger than e-snapshot isolation.

Lemma 17. *If an execution α satisfies du-snapshot isolation then α satisfies e-snapshot isolation, but not vice versa.*

Lemma 18 argues that strict serializability is stronger than snapshot isolation.

Table 1. A quick reference guide showing the relationships between consistency conditions. We remark that a consistency condition determines a set of histories, namely those histories that satisfy the constraints imposed by the condition. Each row and each column of the table represents a consistency condition. Each cell of the table shows the relationship between the consistency condition of the row and the consistency condition of the column that the cell belongs to. For example, the cell that is found in the crossing between the row of e-s (e-serializability) and the column of du-s (du-serializability) contains $\supseteq$. This means that e-s is a superset of du-s, i.e., that e-s is weaker than du-s. The inverse relation is denoted by $\subseteq$, as can be seen in the cell that is found in the crossing between the row of e-ss and the column of e-s: e-ss is stronger than e-s, and thus, it is a subset of e-s. Equality of two conditions is denoted by $=$. Incomparability between them is denoted by $\neq$.

e-ss: e-strict serializability
du-ss: du-strict serializability
e-s: e-serializability
du-s: du-serializability
e-op: e-opacity
du-op: du-opacity
e-cc: e-causal consistency
du-cc: du-causal consistency
e-cs: e-causal serializability
du-cs: du-causal serializability
e-vwc: e-virtual world consistency
du-vwc: du-virtual world consistency
e-svwc: e-strong virtual world consistency
du-svwc: du-strong virtual world consistency
e-si: e-snapshot isolation
du-si: du-snapshot isolation

	e-ss	du-ss	e-s	du-s	e-op	du-op	e-cc	du-cc	e-cs	du-cs	e-vwc	du-vwc	e-svwc	du-svwc	e-si	du-si
e-ss	$=$	$\supseteq$	$\subseteq$	$\neq$	$\supseteq$	$\supseteq$	$\subseteq$	$\neq$	$\subseteq$	$\neq$	$\neq$	$\neq$	$\supseteq$	$\supseteq$	$\subseteq$	$\neq$
du-ss		$=$	$\subseteq$	$\subseteq$	$\neq$	$\supseteq$	$\subseteq$	$\subseteq$	$\subseteq$	$\subseteq$	$\neq$	$\neq$	$\neq$	$\supseteq$	$\subseteq$	$\subseteq$
e-s			$=$	$\supseteq$	$\supseteq$	$\supseteq$	$\subseteq$	$\neq$	$\subseteq$	$\neq$	$\supseteq$	$\supseteq$	$\supseteq$	$\supseteq$	$\neq$	$\neq$
du-s				$=$	$\neq$	$\supseteq$	$\subseteq$	$\subseteq$	$\subseteq$	$\subseteq$	$\neq$	$\supseteq$	$\neq$	$\supseteq$	$\neq$	$\neq$
e-op					$=$	$\supseteq$	$\subseteq$	$\neq$	$\subseteq$	$\neq$	$\subseteq$	$\neq$	$\subseteq$	$\neq$	$\subseteq$	$\neq$
du-op						$=$	$\subseteq$	$\subseteq$	$\subseteq$	$\subseteq$	$\subseteq$	$\subseteq$	$\subseteq$	$\subseteq$	$\subseteq$	$\subseteq$
e-cc							$=$	$\supseteq$	$\supseteq$	$\supseteq$	$\supseteq$	$\supseteq$	$\supseteq$	$\supseteq$	$\neq$	$\neq$
du-cc								$=$	$\neq$	$\supseteq$	$\neq$	$\supseteq$	$\neq$	$\supseteq$	$\neq$	$\neq$
e-cs									$=$	$\supseteq$	$\supseteq$	$\supseteq$	$\supseteq$	$\supseteq$	$\neq$	$\neq$
du-cs										$=$	$\neq$	$\supseteq$	$\neq$	$\supseteq$	$\neq$	$\neq$
e-vwc											$=$	$\supseteq$	$\supseteq$	$\supseteq$	$\neq$	$\neq$
du-vwc												$=$	$\neq$	$\supseteq$	$\neq$	$\neq$
e-swvc													$=$	$\supseteq$	$\subseteq$	$\neq$
du-swvc														$=$	$\subseteq$	$\subseteq$
e-si															$=$	$\supseteq$
du-si																$=$

Lemma 18. *If an execution α satisfies e-strict serializability (du-strict serializability) then α satisfies e-snapshot isolation (du-snapshot isolation), but not vice versa.*

Since strict virtual world consistency and opacity are stronger than strict serializability, Lemma 18 implies that they are stronger than snapshot isolation.

Snapshot isolation is incomparable to virtual world consistency, serializability, causal consistency and causal serializability. For instance, there is an execution which is serializable that does not satisfy snapshot isolation. An example of a history that satisfies snapshot isolation but not serializability is given in Figure 10.

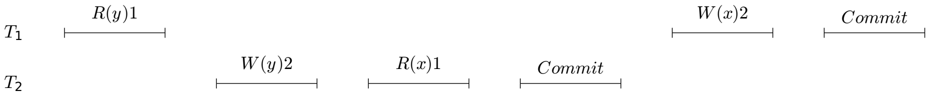

Fig. 10. A history complying with snapshot isolation which is not serializable

Acknowledgments. This work has been supported by the European Commission under the 7th Framework Program through the TransForm (FP7-MC-ITN-238639) project and by the ARISTEIA Action of the Operational Programme Education and Lifelong Learning which is co-funded by the European Social Fund (ESF) and National Resources through the GreenVM project.

We would like to thank Victor Bushkov for his valuable comments in a preliminary version of this chapter and Eleftherios Kosmas for several useful discussions that motivated this work. Many thanks also to Hagit Attiya Petr Kuznetsov, and Sandeep Hans for their comments on a previous version of this article.

References

1. Afek, Y., Avni, H., Dice, D., Shavit, N.: Efficient lock free privatization. In: Lu, C., Masuzawa, T., Mosbah, M. (eds.) OPODIS 2010. LNCS, vol. 6490, pp. 333–347. Springer, Heidelberg (2010)
2. Ahamad, M., Neiger, G., Burns, J.E., Kohli, P., Hutto, P.W.: Causal memory: definitions, implementation, and programming. Distributed Computing 9(1), 37–49 (1995)
3. Ardekani, M.S., Sutra, P., Shapiro, M.: The impossibility of ensuring snapshot isolation in genuine replicated stms. In: TransForm/Euro-TM WTTM 3rd Workshop on the Theory of Transactional Memory, WTTM 2011 (2011)
4. Attiya, H., Hans, S.: Transactions are Back-but How Different They Are? In: 7th ACM SIGPLAN Workshop on Transactional Computing, New Orleans, LA, USA (February 2012)
5. Attiya, H., Hans, S., Kuznetsov, P., Ravi, S.: Safety of deferred update in transactional memory. In: Proceedings of the 33rd International Conference on Distributed Computing Systems, ICDCS 2013, pp. 601–610. IEEE (2013)
6. Attiya, H., Hillel, E., Milani, A.: Inherent limitations on disjoint-access parallel implementations of transactional memory. In: Proceedings of the 21st ACM Symposium on Parallel Algorithms and Architectures, SPAA 2009, pp. 69–78. ACM, New York (2009)
7. Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E., O'Neil, P.: A critique of ansi sql isolation levels. SIGMOD Rec. 24(2), 1–10 (1995)

8. Bernstein, P.A., Hadzilacos, V., Goodman, N.: Concurrency control and recovery in database systems. Addison-Wesley Longman Publishing Co., Inc., Boston (1987)
9. Bushkov, V., Dziuma, D., Fatourou, P., Guerraoui, R.: Snapshot isolation does not scale either. Tech. Rep. TR-437, Foundation of Research and Technology – Hellas (FORTH) (October 2013)
10. Bushkov, V., Dziuma, D., Fatourou, P., Guerraoui, R.: The pcl theorem - transactions cannot be parallel, consistent and live. In: Proceedings of the 4th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2014, pp. 178–187. ACM, New York (2014)
11. Bushkov, V., Guerraoui, R., Kapałka, M.: On the liveness of transactional memory. In: Proceedings of the 31st ACM Symposium on Principles of Distributed Computing, PODC 2012, pp. 9–18. ACM, New York (2012)
12. Dalessandro, L., Spear, M.F., Scott, M.L.: Norec: streamlining stm by abolishing ownership records. In: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2010, pp. 67–78. ACM, New York (2010)
13. Dias, R.J., Seco, J., Lourenço, J.M.: Snapshot isolation anomalies detection in software transactional memory. In: Proceedings of INForum Simpósio de Informática (InForum 2010). Universidade do Minho, Braga (2010)
14. Dice, D., Shavit, N.: What really makes transactions faster? In: 1st ACM SIGPLAN Workshop on Languages Compilers, and Hardware Support for Transactional Computing, TRANSACT 2006 (2006)
15. Doherty, S., Groves, L., Luchangco, V., Moir, M.: Towards formally specifying and verifying transactional memory. Formal Aspects of Computing 25(5), 1–31 (2012)
16. Ellen, F., Fatourou, P., Kosmas, E., Milani, A., Travers, C.: Universal constructions that ensure disjoint-access parallelism and wait-freedom. In: Proceedings of the 31st ACM Symposium on Principles of Distributed Computing, PODC 2012, pp. 115–124. ACM, New York (2012)
17. Guerraoui, R., Kapalka, M.: On obstruction-free transactions. In: Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures, SPAA 2008, pp. 304–313. ACM, New York (2008)
18. Guerraoui, R., Kapalka, M.: On the correctness of transactional memory. In: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2008, pp. 175–184. ACM, New York (2008)
19. Guerraoui, R., Kapalka, M.: Principles of Transactional Memory (Synthesis Lectures on Distributed Computing Theory). Morgan and Claypool Publishers (2010)
20. Harris, T., Larus, J., Rajwar, R.: Transactional Memory, 2nd edn. Morgan and Claypool Publishers (2010)
21. Herlihy, M., Moss, J.E.B.: Transactional memory: architectural support for lock-free data structures. SIGARCH Comput. Archit. News 21(2), 289–300 (1993)
22. 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)
23. Hutto, P., Ahamad, M.: Slow memory: Weakening consistency to enhance concurrency in distributed shared memories. In: Proceedings of the 10th International Conference on Distributed Computing Systems, ICDCS 1990, pp. 302–309. IEEE (1990)
24. Imbs, D., Raynal, M.: Virtual world consistency: A condition for STM systems (with a versatile protocol with invisible read operations). Theoretical Computer Science 444(0), 113–127 (2009), Structural Information and Communication Complexity (SIROCCO) 2009
25. Maessen, J.: Arvind: Store atomicity for transactional memory. Electr. Notes Theor. Comput. Sci. 174(9), 117–137 (2007)
26. Marathe, V.J., Spear, M.F., Scott, M.L.: Scalable techniques for transparent privatization in software transactional memory. In: Proceedings of the 37th International Conference on Parallel Processing (ICPP), pp. 67–74. IEEE Computer Society (2008)

27. Martin, M.M.K., Blundell, C., Lewis, E.: Subtleties of transactional memory atomicity semantics. Computer Architecture Letters 5(2) (2006)
28. Normann, R., Østby, L.T.: A theoretical study of 'snapshot isolation'. In: Proceedings of the 13th International Conference on Database Theory, ICDT 2010, pp. 44–49. ACM, New York (2010)
29. Papadimitriou, C.H.: The serializability of concurrent database updates. Journal of the ACM 26(4), 631–653 (1979)
30. Ramadan, H.E., Roy, I., Herlihy, M., Witchel, E.: Committing conflicting transactions in an stm. In: Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2009, pp. 163–172. ACM, New York (2009)
31. Raynal, M., Thia-Kime, G., Ahamad, M.: From serializable to causal transactions for collaborative applications. In: Proceedings of the 23rd EUROMICRO Conference, EUROMICRO 1997, pp. 314–321. IEEE (1997)
32. Riegel, T., Fetzer, C., Felber, P.: Snapshot isolation for software transactional memory. In: 1st ACM SIGPLAN Workshop on Languages Compilers, and Hardware Support for Transactional Computing, TRANSACT 2006 (2006)
33. Riegel, T., Fetzer, C., Felber, P.: Time-based transactional memory with scalable time bases. In: Proceedings of the 19th ACM Symposium on Parallel Algorithms and Architectures, SPAA 2007, pp. 221–228. ACM, New York (2007)
34. Scott, M.L., Spear, M.F., Dalessandro, L., Marathe, V.J.: Transactions and privatization in delaunay triangulation. In: Proceedings of the 26th ACM Symposium on Principles of Distributed Computing (PODC), pp. 336–337. ACM, New York (2007)
35. Shavit, N., Touitou, D.: Software transactional memory. In: Proceedings of the 14th ACM Symposium on Principles of Distributed Computing, PODC 1995, pp. 204–213. ACM, New York (1995)
36. Siek, K., Wojciechowski, P.T.: Brief announcement: Towards a fully-articulated pessimistic distributed transactional memory. In: Proceedings of SPAA 2013: The 25th ACM Symposium on Parallelism in Algorithms and Architectures, Montreal, Canada, pp. 111–114. ACM (July 2013)
37. Siek, K., Wojciechowski, P.T.: Zen and the art of concurrency control: An exploration of tm safety property space with early release in mind. In: Euro-TM WTTM 6th Workshop on the Theory of Transactional Memory, WTTM 2014 (2014)
38. Spear, M.F., Marathe, V.J., Dalessandro, L., Scott, M.L.: Privatization techniques for software transactional memory. In: Proceedings of the 26th ACM Symposium on Principles of Distributed Computing (PODC), pp. 338–339. ACM, New York (2007)
39. Spear, M.F., Michael, M.M., von Praun, C.: Ringstm: scalable transactions with a single atomic instruction. In: Proceedings of the 20th ACM Symposium on Parallel Algorithms and Architectures, SPAA 2008, pp. 275–284. ACM, New York (2008)
40. Weikum, G., Vossen, G.: Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery. Morgan Kaufmann Publishers (2002)