Model Checking Transactional Memories

Rachid Guerraoui Thomas A. Henzinger Barbara Jobstmann Vasu Singh

EPFL, Switzerland

Abstract

Model checking software transactional memories (STMs) is difficult because of the unbounded number, length, and delay of concurrent transactions and the unbounded size of the memory. We show that, under certain conditions, the verification problem can be reduced to a finite-state problem and we illustrate the use of the method by proving the correctness of several STMs, including two-phase locking, DSTM, TL2, and optimistic concurrency control. The safety properties we consider include strict serializability and abort consistency; the liveness properties include obstruction freedom, livelock freedom, and wait freedom.

Our main contribution lies in the structure of the proofs, which are largely automated and not restricted to the STMs mentioned above. In a first step we show that every STM that enjoys certain symmetry properties either violates a safety or liveness requirement on some program with 2 threads and 2 shared variables, or satisfies the requirement on all programs. In the second step we use a model checker to prove the requirement for the STM applied to a most general program with 2 threads and 2 variables. In the safety case, the model checker constructs a simulation relation between two carefully constructed finite-state transition systems, one representing the given STM applied to a most general program, and the other representing a most liberal STM applied to the same program. In the liveness case, the model checker analyzes fairness conditions on the given STM transition system.

1. Introduction

With the advent of multi-core processors, there is a new urgency for concurrent programming models that give the programmer the illusion of sequentiality and the compiler maximal flexibility. A model that has enjoyed particular recent success is software transactional memory (STM), which allows the programmer to think in coarse-grained code blocks that appear to be executed atomically but does not constrain the compiler by blocking memory access. Inspired by how databases manage concurrency, transactional memory was first introduced by Herlihy and Moss [HM93] in multi-processor design. Later Shavit and Touitou [ST95] introduced STM, a software-based variant of the concept, which enables a new way of looking at concurrent programming. An extensive overview of STM can be found in [LR07]. In this paper, we consider the following STM algorithms: two-phase locking, DSTM [HLMS03], TL2 [DSS06], and optimistic concurrency control [KR81].

Precisely because it encapsulates the difficulty of handling concurrency, the potential of subtle errors in STM implementations is enormous. This makes STM a ripe and important proving ground for formal verification. While there have been initial steps in this direction, the challenge remains daunting for several reasons.

PLDI '08

First, there is no generally agreed upon formal notion of correctness for STM. Scott[Sco06] was the first to provide a formal semantics of STM. However, his weakest correctness criterion requires commit ordering to be preserved. Thus, the popular STM implementation TL2 [DSS06], which does not preserve the commit ordering, falls outside the semantic classification by Scott. Guerraoui and Kapalka [GK08] discussed various alternatives to precisely capture the safety aspect of STM and highlighted the subtle differences with database transactions.

Second, while model checking is the verification technique that is best equipped to find concurrency bugs, model checking is severely handicapped by several sources of unbounded state in STM: memory size, thread count, and transaction length cannot be bounded, and neither can the delay until a transaction commits nor the number of times that a transaction aborts. As with relaxed memory models, special care is needed in formulating a verification problem that is both relevant and solvable, as some problems about sequentializing concurrent systems are undecidable [AMP00].

Third, the specification of an STM universally quantifies over all possible application programs, requiring the desired safety and liveness conditions *for all* programs that are executed on the STM. In this sense, STM verification resembles the problem of checking that a processor implements an instruction set architecture, where the executed programs are also universally quantified. In both cases, the key is to define (and check) a suitable implementation relation [JD94]. While in processor verification, the implementation relation needs to handle pipelines and out-of-order execution, in STM, we need to handle aborted transactions.

We present in this paper a new technique for verifying STM safety and liveness properties. Our technique addresses the three issues above as follows.

First, the safety requirements we consider are *strict serializability* [Pap79] and *abort consistency*. (The latter is a single version read/write restriction of the notion of *opacity* introduced in [GK08].) Strict serializability preserves the order of conflicting operations by transactions, and the order of non-overlapping transactions. Abort consistency ensures, in addition, that aborting transactions do not see an inconsistent state of the memory, which can be disastrous in STMs (due to infinite loops, or exceptions). We study abort consistency, because it provides the programmer with the full sequentiality illusion and, to our knowledge, is satisfied by most STM protocols that claim that illusion [LR07]. Strict serializability is considered here for pedagogical reasons, as it is intuitive and captures the main technical difficulties behind verifying abort consistency. Our verification technique can be extended to the stronger notions of safety discussed by Scott [Sco06] by modifying the semantics of conflict. The liveness requirements we consider are the standard notions of *obstruction freedom*, *livelock freedom*, and *wait freedom* [HLM03, AKH03, Her91].

Second, we exploit the symmetries that are inherent in STM implementations to reduce the unbounded STM state verification to a problem that involves only a small number of threads and shared variables. Specifically, we show that every STM that enjoys certain symmetry properties either violates any of the considered safety and liveness requirements on some program with 2 threads and 2 shared variables, or satisfies the requirement on all programs. The symmetry properties, which expect all threads to be treated equally, are fulfilled by most transactional algorithms, including for in-

stance two-phase locking, DSTM, TL2, and optimistic concurrency control. Similar techniques for reducing unbounded instances of model-checking tasks to small, characteristic instances have been used for verifying protocols with an unbounded number of identical processes [BCG89] and cache-coherence protocols [HQR99].

Third, and perhaps most importantly, we define two finite-state transition systems that generate exactly the strictly serializable (resp. abort consistent) executions of programs with 2 threads and 2 shared variables. These transition systems can be viewed as most liberal *reference STM implementations* guaranteeing strict serializability (resp. abort consistency). To our knowledge, the transition systems presented in this paper provide the first finite-state representation of the language of strictly serializable (resp. abort consistent) executions for transactions that may abort. The finite size of the transition systems is achieved by a careful choice of state, which encompasses for every thread a set of read variables (at most 2), a set of written variables (at most 2), a set of variables not allowed to be read (at most 2), a set of variables not allowed to be written (at most 2), and a set of threads with overlapping, preceding transactions (at most 1). We show that an STM implementation is strictly serializable (resp. abort consistent) iff for a specific, most general program with 2 threads and 2 variables, all executions are permitted by the reference STM implementation. Then, instead of checking language containment between a given STM implementation and the reference implementation, we check for the existence of a simulation relation between both transition systems. (The existence of a simulation relation is a commonly used, efficient sufficient condition for language containment.)

Putting all steps together, we reduce the problem of verifying the safety of an STM implementation, which is unbounded in many dimensions (memory size, thread count, transaction delay, etc.), to a simulation check between two finite-state systems. For two-phase locking, DSTM, TL2, and optimistic concurrency control, we obtain transition systems with up to 4,500 states, and a reference implementation has about 12,500 states. We implemented a simulation checker that automatically verifies strict serializability for optimistic concurrency control and abort consistency for two-phase locking, DSTM, and TL2 in less than 15 minutes. It should be noted that the methodology is applicable to any other STM implementations that fulfill the symmetry properties. Our simulation checker finds that correctness is not self-evident in many STM implementations. To illustrate this, we give an example where reversing two steps in TL2 renders the STM unsafe. In this case, the simulation check provides as counterexample an execution that is not strictly serializable (and thus not abort consistent). We therefore expect our verification tool to be useful to STM designers when they develop or modify STM implementations. Our tool also allows the comparison of different STMs according to whether one allows strictly more executions than another.

On the liveness side, we use again symmetry reduction theorems to check the desired liveness requirement on the finite-state transition system that results from a given STM implementation applied to a most general program with 2 threads and 2 variables. We extend our model checking tool to verify the different liveness properties. In the case of obstruction freedom, this amounts to checking a Streett condition and the check goes through for DSTM. For two-phase locking, TL2, and optimistic concurrency control, the model checker automatically generates counterexamples to obstruction freedom, as it does for DSTM and livelock freedom.

2. Safety in transactional memories

We introduce some notions to define the correctness of a TM. Let V be a set $\{1, \ldots, k\}$ of k *variables*. Let $C = \{\mathsf{commit}\} \cup \{\mathsf{read} \times V\} \cup \{\mathsf{write} \times V\}$ be the set of *commands* on the variables V. Let $T = \{1, \ldots, n\}$ be the set of *threads*. Let $S = C \times T$ be the set of *statements*. We define $\hat{C} = C \cup \{\mathsf{abort}\}$ and $\hat{S} = \hat{C} \times T$. A word $w \in \hat{S}^*$ is a finite sequence of statements. Given a word $w \in \hat{S}^*$, we define the *projection* $w|_t$ of w on thread $t \in T$ as the longest subsequence w' of w such that every statement in w' is in $\hat{C} \times \{t\}$. Given a projection $w|_t = s_0 s_1 \ldots s_m$ of a word w, a statement s_i is *finishing in* $w|_t$ if it is a commit or an abort or the last statement of $w|_t$. A statement s_i is *initiating in* $w|_t$ if it is the first statement in $w|_t$, or the previous statement s_{i-1} is a finishing statement.

Given a projection $w|_t$ of a word w on thread t, a consecutive subsequence $x = s_0 \ldots s_m$ of $w|_t$ is a *transaction* of thread t in w if s_0 is initiating in $w|_t$ and s_m is finishing in $w|_t$, and no other statement in x is finishing in $w|_t$. The transaction x is *committing* in w if s_m is a commit statement. The transaction x is *aborting* in w if s_m is an abort statement. Otherwise, the transaction x is *pending* in w. Given a word w and two transactions x and y in w (possibly of different threads), we say that x *precedes* y in w, written as $x <_w y$, if the finishing statement of x occurs before the initiating statement of y in w. A word w is *sequential* if for every pair (x, y) of transactions in w, either $x <_w y$ or $y <_w x$.

We define a function $com : \hat{S}^* \to S^*$ such that for all words $w \in \hat{S}^*$, the word $com(w)$ is the longest subsequence w' of w such that every statement in w' is part of a committing transaction in w. Thus, $com(w)$ consists of all statements of all committing transactions in w.

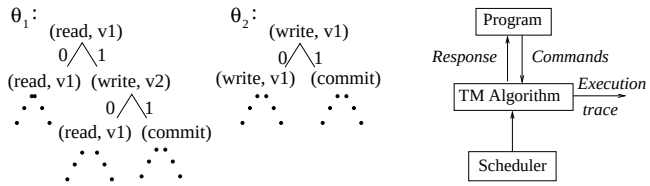
A transaction x of a thread t *writes* to a variable v if x contains a statement $((\mathsf{write}, v), t)$. A statement $s = ((\mathsf{read}, v), t)$ in x is a *global read* of a variable v if there is no statement $((\mathsf{write}, v), t)$ before s in the transaction x. A transaction x of a thread t *globally reads* a variable v if there exists a global read of variable v in transaction x. A word w is *transaction equivalent* to a word w' if for every thread $t \in T$, we have $w|_t = w'|_t$.

2.1 Safety criteria

Conflict serializability (cf. [EGLT76]) is a commonly used correctness criterion for concurrent systems and, in particular, for transactional systems. Conflict serializability allows us to omit the values of read and write commands, since the consistency of the values follows from preserving the order of conflicts. In the context of transactional memories, a stronger property, called strict serializability, is considered. Strict serializability preserves the order of non-overlapping transactions. We note that strict serializability does not state any restrictions on the operations of the aborting transactions. In the scope of STMs, a stronger notion of correctness, referred to as *abort consistency* has been suggested [GK08, HLMS03] to avoid unexpected side effects, like infinite loops, or array bound violations. Abort consistency requires that a word is strictly serializabile, and that the aborting transactions do not see read inconsistent values.

Now, we formalize these correctness criteria. We start with the notion of a conflict. Transactional memories use direct update semantics (every transaction modifies the shared variables in place and restores them upon abort), or deferred update semantics (every transaction modifies a local copy, and changes the shared copy upon a commit). We choose to define conflicts under the deferred update semantics. All our work can be similarly applied to TMs with direct update semantics, with a slight modification of definition of conflict. A statements s_1 of transaction x and a statement s_2 of transaction y ($x \neq y$) *conflict* in a word w if (i) s_1 is a global read of variable v and s_2 is a commit and y writes to v, or (ii) s_1 and s_2 are both commits, and x and y write to v.

A word $w = s_0 \ldots s_n$ is *conflict equivalent* to a word w' if (i) w is transaction equivalent to w', and (ii) for every pair s_i, s_j of statements in w, if s_i and s_j conflict and $i < j$, then s_i occurs before s_j in w'. A word $w = s_0 \ldots s_m$ is *strictly equivalent* to a



(a) An example program on two threads and two variables

(b) Interaction in the system

Figure 1. Our framework of transactional memory

word w' if (i) w is conflict equivalent to w' and (ii) for every pair x, y of transactions in w, if $x <_w y$ then $x <_{w'} y$.

A word $w \in \hat{S}^*$ is *strictly serializable* if there exists a sequential word w' such that w' is strictly equivalent to $com(w)$. Furthermore, we define that a word w is *abort consistent* if there exists a sequential word w' such that w' is strictly equivalent to w. (Note that w may contain pending transactions.) We note that given a word w, if w is abort consistent, then w is strictly serializable.

Example. Consider $w =$ ((read, v_1), t_1), ((write, v_1), t_2), ((write, v_2), t_2), (commit, t_2), ((read, v_2), t_1). w has two transactions: (i) a pending transaction ((read,v_1),t_1), ((read, v_2),t_1), and (ii) a committing transaction ((write, v_1), t_2), ((write, v_2), t_2), (commit, t_2). The following pairs of statements conflict: (((read, v_1), t_1),(commit, t_2)) and (((read, v_2), t_1), (commit, t_2)). The word w is strictly serializable because $com(w)$ consists only of ((write, v_1), t_2), ((write, v_2), t_2), and (commit, t_2). On the other hand, w is not abort consistent since t_1 reads the old value of v_1 (before t_2 commits) and the new value of v_2 (committed by t_2).

2.2 Transactional memories

We consider a thread as our basic sequential unit of computation, and describe a program to be a collection of threads. Each thread consists of a sequence of transactions. In our formalism, we allow programs to retry a transaction, or start another transaction on an abort. Formally, an *unrolled thread* θ on C is a function $\theta : \mathbb{B}^* \to C$. We write Θ for the set of unrolled threads. Defining unrolled threads as infinite binary trees on commands makes the representation independent of specific control flow statements, such as exceptions for handling abort. For every command of the thread, we define two successor commands, one if the command is successfully executed, and another if the command fails due to an abort of the transaction.

Note that this definition allows us to capture easily different retry mechanisms of TMs, e.g., retry the same transaction until it succeeds or try other transaction after an abort. We define a *program* p on n threads and k variables as an n-tuple $p = \langle \theta^1, \ldots, \theta^n \rangle$ of unrolled threads on C. Figure 1(a) shows an example program on two threads and two variables. Let $P^{n,k}$ be the set of all programs on n threads and k variables. Let P be the set of all programs.

We define a transactional memory as an abstract function that takes as input a program, and produces a set of infinite words. Formally, a *transactional memory* is a function $M : P \to 2^{\hat{S}^\omega}$. A transactional memory M *ensures strict serializability for all programs with n threads and k variables* if for every program $p \in P^{n,k}$, for every word $w \in M(p)$, all finite prefixes of w are strictly serializable. Similarly, M *ensures abort consistency for all programs with n threads and k variables* if for every program $p \in P^{n,k}$, for every word $w \in M(p)$, all finite prefixes of w are abort consistent. We say that a transactional memory M *ensures strict serializability* (resp. *abort consistency*) if it ensures strict serializability (resp. abort consistency) for all programs with arbitrary number n of threads and arbitrary number k of variables.

3. Transactional memory algorithms

We use state transition systems to define TMs. A TM algorithm is a family of TM transition systems, one for n threads and k variables, for every n and k. The TM transition system consists of a set of states, an initial state, a transition relation between the states, and an extended set of commands depending on the underlying TM. For example, a given TM may require that a thread locks a variable before writing to the variable, or that a thread validates the variables read in a transaction, before accessing a new variable.

A TM algorithm interacts with a program and a scheduler (see Fig. 1(b)). The scheduler chooses a thread and determines the next command of that thread to be executed. The TM transition system decides whether the command can be executed in a single atomic step, or in several atomic steps, or has to be aborted. Given a program, a scheduler, and a TM transition system, we get an execution trace. Projecting this trace to the set $\hat{C}$ of commands, we get a word in $\hat{S}^*$. We describe the language of a TM transition system as the set of words on $\hat{S}^*$ that it can produce.

Formally, a *scheduler* σ on T is a function $\sigma : \mathbb{N} \to T$. Let Σ be the set of schedulers. We define a *TM algorithm* A as a family of *TM transition systems* $A^{n,k} = \langle Q, q_{init}, D, \delta \rangle$ for each n and k, where Q is a set of states, q_{init} is the initial state, D is the set of extended commands with $C \subseteq D$, and $\delta \subseteq Q \times C \times \hat{S}_D \times Resp \times Q$ is the deterministic or non deterministic transition relation, where $\hat{S}_D = (D \cup \{\mathsf{abort}\}) \times T$ and $Resp = \{0, 1, \bot\}$. The transition relation δ obeys the following rules:

1. if there exists a transition $(q, c, (d, t), r, q_1) \in \delta$ of thread t to state $q_1 \in Q$ such that $r = \bot$, then for every transition $(q_1, c_1, (d_1, t_1), r_1, q_2) \in \delta$ with $t_1 = t$, we have $c_1 = c$. In this case, we say that only command c_1 is *enabled* in q_1 for thread t.

2. if for all transitions $(q, c, (d, t), r, q_1) \in \delta$ of thread t to state $q_1 \in Q$, $r \neq \bot$ holds then there exists for every command $c_1 \in C$ a transition $(q_1, c_1, (d_1, t), r_1, q_2) \in \delta$. In this case, we say that every command $c_1 \in C$ is *enabled* in q_1 for thread t.

3. for all $q \in Q$ and for all transitions $(q, c, (d, t), r, q_1) \in \delta$ such that $d = \mathsf{abort}$, we have $r = 0$.

4. for all $q \in Q$ and $(c, t) \in S$, if c is not enabled in q for thread t then there exists a transition $(q, c, (d, t), r, q_1) \in \delta$ with $d = \mathsf{abort}$ for some $q_1 \in Q$. In this case, we say that the command c is *abort enabled* in the state q for thread t.

Moreover, for a deterministic transition relation δ, we have for all $q \in Q$ and $(c, t) \in S$, if $(q, c, (d_1, t), r_1, q_1) \in \delta$ and $(q, c, (d_2, t), r_2, q_2) \in \delta$ then $d_1 = d_2$, $q_1 = q_2$, and $r_1 = r_2$. Unless otherwise stated, TM transition systems have deterministic transition relations.

Let $p = \langle \theta^1, \ldots, \theta^n \rangle$ be a program in $P^{n,k}$. Let σ be a scheduler on n threads. A *run* $\rho = \langle q_0, l_0 \rangle \langle q_1, l_1 \rangle \ldots$ of $A^{n,k}$ with scheduler σ on program p is an infinite sequence of states together with program locations, where $l_j = \langle l_j^1, \ldots, l_j^n \rangle \in (\mathbb{B}^*)^n$ for all $j \geq 0$ and (i) $q_0 = q_{init}$ and $l_0 = \langle \epsilon, \ldots, \epsilon \rangle$, (ii) for all $j \geq 0$ there exists a transition $(q_j, c_j, (d_j, t_j), r_j, q_{j+1}) \in \delta$ such that $t_j = \sigma(j)$ and $c_j = \theta^{t_j}(l_j^{t_j})$ and for all $t \in T$ if $t \neq t_j$ or $r_j = \bot$ then $l_{j+1}^t = l_j^t$, otherwise $l_{j+1}^t = l_j^t \cdot r_j$. We associate with ρ an *execution trace* $s_0 s_1 \ldots$ in $\hat{S}_D^\omega$ such that $s_j = (d_j, t_j)$ for all $j \geq 0$. We define the *language* $L(A^{n,k})$ of $A^{n,k}$ as the set of all finite words $w \in \hat{S}^*$ such that $w = e|_{\hat{S}}$, where e is a finite prefix of an execution trace of $A^{n,k}$ for some program p on n threads and k variables, and some scheduler σ on n threads.

A TM algorithm A defines a transactional memory M such that for all n, k, for every program p in $P^{n,k}$ and every word $w \in \hat{S}^\omega$, we have $w \in M(p)$ iff there exists a scheduler σ on T such that $w = e|_{\hat{S}}$, where e is the execution trace of p and σ on the TM algorithm A. It follows that a TM M defined by a TM algorithm A ensures strict serializability (abort consistency) for all programs with n threads and k variables iff all words in $L(A^{n,k})$ are strictly serializable (abort consistent).

In the following sections, we describe different transactional memories as TM algorithms. To simplify the description, we view a state q of the corresponding TM transition systems as an n-tuple $\langle q^1 \dots q^n \rangle$, where each component q^t corresponds to a thread t and is called *thread state* of t.

3.1 The sequential TM

To keep our first example simple, we describe a sequential TM. The sequential TM executes the transactions sequentially (as ideally suited for a uniprocessor). We define the sequential TM M_{seq} using a sequential TM algorithm A_{seq}. The sequential TM transition system $A^{n,k}_{seq}$ for n threads and k variables is given by the tuple $\langle Q, q_{init}, D, \delta \rangle$. The thread state q^t of thread t is $\{0, 1\}$. The initial state $q_{init} = \langle 0, \dots, 0 \rangle$. The set of extended commands is $D = C$. A transition $(q_1, c, (d, t), r, q_2)$ is in δ if c is enabled in q_1 for thread t and one of the following holds:
1. Read/Write. (i) $c \in \{\mathsf{read}, \mathsf{write}\} \times V$ and $d = c$ and $r = 1$, and (ii) $q_1^u = 0$ for all $u \neq t$ and $q_2^t = 1$, and (iii) $q_2^u = q_1^u$ for all $u \neq t$ (When a thread reads or writes a variable, the state of all other threads should be false. The state of t is set to true.)
2. Commit. (i) $c = \mathsf{commit}$ and $d = c$ and $r = 1$, and (ii) $q_1^u = 0$ for all $u \neq t$, and (iii) $q_2^t = 0$ and $q_2^u = q_1^u$ for all $u \neq t$ (When a thread t commits, the state of all other threads should be false. The state of t is set to false.)

A transition $(q_1, c, (\mathsf{abort}, t), 0, q_2)$ is in δ if c is abort enabled in q_1 for thread t and $q_2 = q_1$.

3.2 The two-phase locking TM

Our second example of a TM algorithm is based on two phase locking (2PL) protocol, commonly used in database transactions. Every transaction locks the variables it reads or writes before accessing them, and releases all the acquired locks during the commit. We define the 2PL TM M_{2PL} using a 2PL TM algorithm A_{2PL}. The 2PL TM transition system $A^{n,k}_{2PL}$ for n threads and k variables is given by the tuple $\langle Q, q_{init}, D, \delta \rangle$. The thread state q^t of thread t is a subset of V. It denotes the variables locked by the thread. The initial state $q_{init} = \langle \emptyset, \dots, \emptyset \rangle$. The set of extended commands is $D = C \cup (\{\mathsf{lock}\} \times V)$. δ is the transition relation such that $(q_1, c, (d, t), r, q_2) \in \delta$ if c is enabled in q_1 for thread t and one of the following holds:
1. Read/Write. (i) $c \in \{\mathsf{read}, \mathsf{write}\} \times \{v\}$ and $d = c$ and $r = 1$, and (ii) $v \in q_1^t$, and (iii) $q_2 = q_1$ (When a thread has to read or write v and it already holds a lock on v, the read or write is executed by the TM.)
2. Lock. (i) $c \in \{\mathsf{read}, \mathsf{write}\} \times \{v\}$ and $d = (\mathsf{lock}, v)$ and $r = \perp$, and (ii) $v \notin q_1^t$ and for all $u \neq t$, we have $v \notin q_1^u$, and (iii) $q_2^t = q_1^t \cup \{v\}$, and (iv) $q_2^u = q_1^u$ for all $u \neq t$ (When a thread has to read or write v, and it does not hold a lock on v, the thread first locks v.)
3. Commit. (i) $c = \mathsf{commit}$ and $d = c$ and $r = 1$, and (ii) $q_2^t = \emptyset$, and (iii) $q_2^u = q_1^u$ for all threads $u \neq t$ (When a thread commits, it releases all the locks.)

A transition $(q_1, c, (\mathsf{abort}, t), 0, q_2)$ is in δ if c is abort enabled in q_1 for thread t and $q_2^t = \emptyset$ and $q_2^u = q_1^u$ for all threads $u \neq t$.

3.3 The dynamic software transactional memory

Dynamic software TM (DSTM) [HLMS03] is one of the most popular STM algorithms. The algorithm exists in many flavors. In this work, we focus on one of them, called *invisible read DSTM*, where the transactions require ownership of variables only for writing. The reads are not visible to the writers. Upon reading, the transactions validate their read set. In our work, we ignore optimizations like early release possible in DSTM. Our TM transition system does not directly allow one thread to abort another thread. So, we allow a thread to set an abort flag for another thread and change the state of the aborted thread appropriately, and also, require that a thread aborts whenever the abort flag is set for the thread. We define the DSTM TM M_{dstm} using a DSTM TM algorithm A_{dstm}. The DSTM TM transition system $A^{n,k}_{dstm}$ for n threads and k variables is given by $\langle Q, q_{init}, D, \delta \rangle$. A thread state q^t of thread t is defined as a 3-tuple $\langle status^t, rs^t, os^t \rangle$, where $status^t \in \{\mathsf{aborted}, \mathsf{valid}, \mathsf{invalid}\}$ is the status of thread i, $rs^t \subseteq V$ is the read set of thread i, and $os^t \subseteq V$ is the ownership set of thread i. For every thread, the initial thread state of thread t is $q^t_{init} = \langle \mathsf{valid}, \emptyset, \emptyset \rangle$. The set of extended commands is $D = C \cup (\{\mathsf{own}\} \times V)$. A transition $(q_1, c, (d, t), r, q_2)$ is in δ if c is enabled in q_1 for thread t and one of the following holds:
1. Local read. (i) $c = (\mathsf{read}, v)$ and $d = c$ and $r = 1$, and (ii) $v \in os_1^t$ and $status_1^t \neq \mathsf{aborted}$, and (iii) $q_2 = q_1$ (When a thread read v such that the read is not global, nothing changes)
2. Global read. (i) $c = (\mathsf{read}, v)$ and $d = c$ and $r = 1$, and (ii) $v \notin os_1^t$ and $status_1^t = \mathsf{valid}$, and (iii) $rs_2^t = rs_2^t \cup \{v\}$, and $os_2^t = os_1^t$ and $status_2^t = \mathsf{valid}$, and (iv) $q_2^u = q_1^u$ for all threads $u \neq t$ (When a thread reads v globally, the status of the thread should be valid and v is added to the read set of the thread)
3. Own. (i) $c = (\mathsf{write}, v)$ and $d = (\mathsf{own}, v)$ and $r = \perp$, and (ii) $status_1^t \neq \mathsf{aborted}$, and (iii) $rs_2^t = rs_1^t$ and $os_2^t = os_1^t \cup \{v\}$ and $status_2^t = status_1^t$, and (iv) for all threads $u \neq t$ if $v \in os_1^u$ then $status_2^u = \mathsf{aborted}$, and $os_2^u = \emptyset$, and $rs_2^u = \emptyset$, otherwise $status_2^u = status_1^u$, $os_2^u = os_1^u$, and $rs_2^u = rs_1^u$. (When a thread writes to v, it should first own v, the status should not be aborted, the variable v is added to the owned set of the thread. If v was owned by some other thread earlier, the status of that thread is aborted and its read and own sets are set to empty).
4. Write. (i) $c = (\mathsf{write}, v)$ and $d = c$ and $r = 1$, and (ii) $status_1^t \neq \mathsf{aborted}$ and $v \in os_1^t$, and (iii) $q_2^u = q_1^u$ for all $u \in T$ (A thread can write to v if the status is not aborted and the variable v is in the own set of the thread).
5. Commit. (i) $c = \mathsf{commit}$ and $d = c$ and $r = 1$, and (ii) $status_1^t = \mathsf{valid}$, and (iii) $os_2^t = \emptyset$, and $rs_2^t = \emptyset$, and (iv) for all threads $u \neq t$, $rs_2^u = rs_1^u$, $os_2^u = os_1^u$, and $status_2^u = \mathsf{invalid}$ if $rs_1^u \cup os_1^t \neq \emptyset$ and $status_2^u = status_1^u$ otherwise. (A thread t commits if the status is valid. The own and read sets of the thread are set to empty. The status of threads whose read set intersects with the own set of t is set to invalid.)

A transition $(q_1, c, (\mathsf{abort}, t), 0, q_2)$ is in δ if the command c is abort enabled in q_1 for thread t, and $status_2^t = \mathsf{valid}$, and $rs_2^t = \emptyset$ and $os_2^t = \emptyset$, and $q_2^u = q_1^u$ for all threads $u \neq t$.

3.4 The TL2 transactional memory

Transactional locking 2 (TL2) [DSS06] is a TM which works as follows. First, a transaction reads and writes locally to the variables. After the transaction has locally completed, the thread acquires locks for the variables it writes to. Then, the transaction is validated using version numbers. If for all the variables in the read set, the version is consistent, and no other thread owns the variable, then the transaction is allowed to commit. We note that TL2 uses locks for synchronization and version control to check validation. A version number is maintained for every variable, which is incremented

when the variable is written. Every transaction reads the variable along with the version number. A transaction successfully commits if all the variables that it reads have the same version number at the time of commit. TL2 uses version control to validate the read set efficiently in a distributed setting. To model TL2 using a finite state TM transition system, we replace the version control by invalidation. When a transaction commits, it invalidates the transactions whose read set intersects with the transaction's write set.

We define the TL2 TM M_{TL2} using the TL2 TM algorithm as A_{TL2}. The TL2 TM transition system $A^{n,k}_{TL2}$ for n threads and k variables is given by the tuple $\langle Q, q_{init}, D, \delta\rangle$. A thread state q^t of thread t in the TL2 algorithm is defined as a 4-tuple $\langle status^t, rs^t, ws^t, ls^t\rangle$, where $status^t \in \{\mathsf{valid}, \mathsf{invalid}, \mathsf{validated}, \mathsf{commitrdy}\}$ $rs^t \subseteq V$ is the read set, $ws^t \subseteq V$ is the write set, and $ls \subseteq V$ is the lock set. The initial thread state $q^t_{init} = \langle \mathsf{valid}, \emptyset, \emptyset, \emptyset\rangle$ for all threads $t \in T$. The set of extended commands is $D = C \cup (\{\mathsf{lock}\} \times V) \cup \{\mathsf{validate}, \mathsf{chklock}\}$. We express the transition relation informally. The formal transition relation can be obtained, as in the previous examples. A transition on a command c for a thread t in state q occurs if c is enabled in the state q for thread t, where the command is one of the following:

1. Local read. A thread can read v if the read is local.

2. Global read. When a thread reads v and the read is global, the status of the thread should be valid, the lock set should be empty, and the variable v is added to the read set.

3. Lock. When a thread commits, the thread first locks every variable in the write set. The status should be valid or invalid. No other thread should hold the lock on the variable, and the variable is added to the lock set of the thread.

4. Write. When a thread writes to v, the status should be valid or invalid. The lock set of the thread should be empty. The variable v is added to the write set of the thread.

5. Validate. When a thread has to commit, it validates the read set after acquiring the locks on all the variables in the write set. The status of the thread should be valid, and it is set to validated.

6. Chklock. When a thread has to commit, after validating the read set, it is checked that the read set of the thread does not intersect with the lock set of any other thread. If so, the status is set to commitrdy, and the thread can now successfully commit.

7. Commit. When a thread has to commit, if the status is commitrdy, the thread commits.

A thread t aborts on a command c in state q if c is abort enabled in state q for thread t. The read set, lock set, and the write set are changed to empty, and the status is set to valid.

3.5 The optimistic concurrency control TM

We now discuss a common concurrency protocol used in databases. It was proposed by Kung et al. [KR81] and called optimisitic concurrency control (OCC). The OCC TM executes the transactions of the threads without any synchronization. Before committing, every transaction chooses a sequence number and validates its read set. Transactions commit in the order of sequence numbers.

We define the OCC TM M_{occ} using an OCC TM algorithm A_{occ}. We refer to the OCC TM transition system with n threads and k variables as $A^{n,k}_{occ}$. The formal definition of the transition system can be obtained from the original algorithm, as we did in the previous examples.

Table 1 shows execution traces and words for the example program in Figure 1(a) and different schedulers with every transaction memory described above.

4. Reduction theorems for safety

We present two reduction theorems, corresponding to strict serializability and abort consistency. These theorems state that if a TM ensures strict serializability (abort consistency) for all programs on two threads and two variables then the TM ensures strict serializability (abort consistency). The reduction theorems rely on certain symmetry properties of transactional memories. These properties are satisfied by all TMs that were discussed in the previous section.

Table 1. Examples of execution traces and words in the language of different TM algorithms. Notation: r = read, w = write, c = commit, a = abort, l = lock, o = own, v = validate, cl = chklock, s = serialize. Command (c, t) is written as c_t.

TM	Scheduler output	1st trace: Execution trace e 2nd trace: $w = e\|_{\hat{S}}$ in $L(A)$
seq	11122...	$(r,1)_1, (w,2)_1, c_1, (w,1)_2, c_2 \ldots$ $(r,1)_1, (w,2)_1, c_1, (w,1)_2, c_2$
	112122...	$(r,1)_1, (w,2)_1, a_2, c_1, (w,1)_2, c_2 \ldots$ $(r,1)_1, (w,2)_1, a_2, c_1, (w,1)_2, c_2$
2PL	111112...	$(l,1)_1, (r,1)_1, (l,2)_1, (w,2)_1, c_1, (l,2)_2 \ldots$ $(r,1)_1, (w,2)_1, c_1$
	1211112...	$(l,1)_1, a_2, (r,1)_1, (l,2)_1, (w,1)_1, c_1, (l,2)_2 \ldots$ $a_2, (r,1)_1, (w,2)_1, c_1$
dstm	1221112...	$(r,1)_1, (o,1)_2, (w,1)_2, (o,2)_1, (w,2)_1, c_2, c_1 \ldots$ $(r,1)_1, (w,1)_2, (w,2)_1, c_2, c_1$
	1222111...	$(r,1)_1, (o,1)_2, (w,1)_2, c_2, (o_2)_1, (w,2)_1, a_1 \ldots$ $(r,1)_1, (w,1)_2, c_2, (w,2)_1, a_1$
TL2	11211122212...	$(r,1)_1, (w,2)_1, (w,1)_2, (l,2)_1, v_1, cl_1, (l,1)_2, v_2, cl_2, c_1, c_2 \ldots$ $(r,1)_1, (w,2)_1, (w,1)_2, c_1, c_2$
	1121211222...	$(r,1)_1, (w,2)_1, (w,1)_2, (l,2)_1, (l,1)_2, v_1, a_1, v_2, cl_2, c_2 \ldots$ $(r,1)_1, (w,2)_1, (w,1)_2, a_1, c_2$
occ	1211212...	$(r,1)_1, (w,1)_2, (w,2)_1, s_1, s_2, c_1, c_2 \ldots$ $(r,1)_1, (w,1)_2, (w,2)_1, c_1, c_2$
	1221112...	$(r,1)_1, (w,1)_2, s_2, (w,2)_1, s_1, a_1, c_2 \ldots$ $(r,1)_1, (w,1)_2, (w,2)_1, a_1, c_2$

We define four symmetry properties for TMs. Let M be a transactional memory. Let p be a program on n threads and k variables. Let w be a finite prefix of a word in $M(p)$.

P1. *Symmetry in threads.* Let w have no aborting transactions and let X be the set of committed transactions of thread t in w. Let there exist a thread u such that for all committed transactions y of u and all committed transactions $x \in X$, either $x <_w y$ or $y <_w x$. Then the word w' obtained by renaming all transactions of thread u to be from thread t is a finite prefix of a word in $M(p')$ for some program p' on $n-1$ threads and k variables.

Example. Let w = ((read, v_1), t_2), (commit, t_2), ((write, v_1), t_1), (commit, t_1), ((write, v_2), t_2), (commit, t_2). Then, the word w' = ((read, v_1), t_2), (commit, t_2), ((write, v_1), t_2), (commit, t_2), ((write, v_2), t_2), (commit, t_2) is a finite prefix of a word in $M(p')$ for some program p'.

P2. *Transaction projection.* Let X be the set of transactions in w. We define a *transaction projection* of w on $X' \subseteq X$ as the longest subsequence of w such that all the statements are from transactions in X'. This property states that the transaction projection of w on X' where X' is a subset of the set of committing and pending transactions in w is in $M(p')$ for some program p'. Note that if we project the word on part of the aborting transactions, then the resulting word is not guaranteed to be in $M(p')$ for any program p'.

Example. Let w = ((read, v_1), t_3), ((write, v_2), t_3), ((write, v_1), t_1), (commit, t_1), ((write, v_2), t_2), (commit, t_2). Then, the word w' = ((write, v_1), t_1), (commit, t_1), ((write, v_2), t_2), (commit, t_2) is in $M(p')$ for some program p'.

P3. *Variable projection.* Let w have no aborting transactions. We define a *variable projection* of w on $V' \subseteq V$ as the longest subsequence of w such that all the statements are reads or writes to variables in V' or commit or abort statements. Given a program p, we define the variable projection of p on $V' \subseteq V$ as the program

obtained by removing all reads and writes statements to variables in $V \setminus V'$ from all unrolled threads in p. This property states that the variable projection of w on $V' \subseteq V$ is in $M(p')$, where p' is the projection of p on the variables V'.

Example. Let w be the word as in the example of property P2. The word $w' = ((\mathsf{write}, v_2), t_3), ((\mathsf{write}, v_2), t_2), (\mathsf{commit}, t_2)$ is in $M(p')$ for the p', where p' is the projection of p on $\{v_2, v_3\}$.

P4. *Monotonicity property for strict serializability (abort consistency).* This property states that for a class of words, if a word is produced by a TM, then more sequential versions of the word are also produced by the TM. Formally, let the word $w_p \in \hat{S}^*$ be strictly serializable (abort consistent) and let only transaction x be pending in w_p. We define W to be the set of words w' such that $w' = w_p \cdot s$ where s is a statement of x and s is not an aborting statement. We call W the set of *extensions of* w_p. If w (a finite prefix of a word in $M(p)$) is extension of some word w_1 with the same properties as w_p and $w = w_1 \cdot s$ then there exists a word w_2 that is strictly equivalent to w_1 such that $com(w_2)$ is sequential and $w_2 \cdot s$ is also a finite prefix of a word in $M(p)$.

Example. Let $w = ((\mathsf{read}, v_1), t_1), ((\mathsf{read}, v_1), t_3), ((\mathsf{write}, v_1), t_2), ((\mathsf{write}, v_2), t_2), ((\mathsf{read}, v_2), t_3), (\mathsf{commit}, t_2), (\mathsf{commit}, t_3), ((\mathsf{read}, v_2), t_1), (\mathsf{commit}, t_1)$. The word w is an extension of word w_1. Then, the word $w_2 \cdot s = ((\mathsf{read}, v_1), t_3), ((\mathsf{read}, v_2), t_3), (\mathsf{commit}, t_3), ((\mathsf{read}, v_1), t_1), ((\mathsf{write}, v_1), t_2), ((\mathsf{write}, v_2), t_2), (\mathsf{commit}, t_2), ((\mathsf{read}, v_2), t_1), (\mathsf{commit}, t_1)$ is a finite prefix of a word in $M(p)$.

Theorem 1. Let M be a TM that satisfies the properties P1, P2, P3, and P4. Moreover, M ensures strict serializablity (resp. abort consistency) for all programs on two threads and two variables. Then the TM M ensures strict serializability (resp. abort consistency).

Proof. We prove the theorem for strict serializability. A similar proof holds for abort consistency. The proof is by contradiction. Let p be a program in $P^{n,k}$. Let w be a finite prefix of a word in $M(p)$ such that w is not strictly serializable. Let w_p be the longest prefix of w such that w_p is strictly serializable and let $w_1 = w_p \cdot s$, where $s = (c, t)$ is a statement of transaction x. Let X be the set of committed transactions in w_p. By property P2, there exists a word w_2 generated by projecting w_1 to $X \cup \{x\}$ such that w_2 is a finite prefix of a word in $M(p_2)$ for some program p_2. We note that $w_2 = w'_p \cdot s$ and w'_p is strictly serializable and w_2 is not strictly serializable. So, using property P4, there exists a word w''_p that is strictly equivalent to w'_p such that $com(w''_p)$ is sequential and the word $w_3 = w''_p \cdot s$ is a finite prefix of a word in $M(p_2)$. In w_3 only one transaction, x, does not execute sequentially. Using property P1, we rename the threads for the transactions in w_3. We let all transactions except x to be executed by thread u. Let this renaming give word w_4. We note that the last statement of x is a commit. As w_4 is not strictly serializable, we know (by the definition of conflict) that one of the following holds: (i) $s_1 = ((\mathsf{read}, v_1), t)$ and $s_2 = ((\mathsf{read}, v_2), t)$ are global reads of transaction x such that some transaction y of thread u writes to v_1 and some transaction y' of u with $y' = y$ or $y <_{w_4} y'$ writes to v_2 and both commit between s_1 and s_2, (note that y and y' cannot overlap due to the structure of w_4,) or (ii) $s_1 = ((\mathsf{read}, v_1), t)$ is a global read of transaction x such that some transaction y of thread u writes to v_1 and commits after s_1, and there is a committing transaction y' with $y' = y$ or $y <_{w_4} y'$ which has a command (read, v_2) or (write, v_2), and x also writes to v_2. (Note that v_1 may be same as v_2). Let w_5 be a variable projection of w_4 on $\{v_1, v_2\}$. We know that w_5 is a finite prefix of a word in $M(p_5)$ for some program p_5 on two threads and two variables, by property P3. Also, we note that w_5 is not strictly serializable. As M ensures strict serializability for all programs on two threads and two variables, we get a contradiction. Thus, there is no such program p_5. This leads us to a contradiction. □

5. The reference TM algorithms

To verify the safety properties of a transactional memory, we take the following approach. We construct a reference TM algorithm for strict serializability (RSS TM algorithm), which has an execution trace for every strictly serializable word. Similarly, we construct a reference TM algorithm for abort consistency (RAC TM algorithm), which has an execution trace for every abort consistent word. Then, we show that a given TM M defined by a TM algorithm A ensures strict serializability (resp. abort consistency) iff all words in $L(A^{2,2})$ are in the language of the RSS (RAC) TM transition system for two threads and two variables (due to the reduction theorems).

The key insight that makes our technique work is that the reference TM algorithms for strict serializability and abort consistency for two threads and two variables can be defined as ***finite-state*** transition systems. This is not obvious, as threads may be delayed arbitrarily, transactions may contain arbitrarily many instructions and may be aborted arbitrarily often. We present the RSS TM transition system first, because it provides the basis for defining the RAC TM transition system. Suitable finite-state reference TM transition systems can also be defined for stronger notions of safety, such as those used by Scott [Sco06], by modifying the semantics of conflict.

5.1 The reference TM algorithm for strict serializability

The classical approach to checking whether a word is strictly serializable is to construct a directed graph $G = (V, E)$ (called the conflict graph [Pap79]) of the committing transactions in the word. The conflict graph captures the precedence of the committing transactions based on the conflicts. Given a word $w = s_0 s_1 \ldots$, the transactions in w form the set V of vertices in the conflict graph. There exists an edge from a vertex v_1 to a vertex v_2 if v_2 finishes before v_1 starts, or a statement s_i of v_1 conflicts with a statement s_j of v_2 and $i > j$. The conflict graph G is acyclic iff the word w is strictly serializable. We note that the size of this construction is unbounded. The following parametrized word illustrates the point. $w_n = ((\mathsf{read}, v_1), t_1), (((\mathsf{write}, v_1), t_2), (\mathsf{commit}, t_2))^n, (\mathsf{commit}, t_1)$. The number of vertices in the conflict graph of w_n is $n + 1$. Thus, we cannot aim to create a finite transition system for the RSS TM algorithm using conflict graphs. We provide a novel approach to check whether a word is strictly serializable or not. In our knowledge, this is the first finite state representation for the language of strictly serializable words, when transactions may abort. The idea of maximal serializability was earlier addressed in a restricted scope [FR85] for a bounded number of non-aborting transactions with a bounded number of instructions per transaction. The idea was built upon a notion of transitive conflicts, which does not hold when transactions may abort.

The key idea to get around the problem of infinite states is to maintain sets called *prohibited read and write sets* for every thread. These sets allow us to handle unbounded delay between transactions, as committing transactions store the required information in these sets of other threads. Once a transaction commits or aborts, we need not remember it (unlike conflict graphs). Thus, we need to store information of at most one transaction per thread. The RSS TM transition system is based on the following notion: *Every committing transaction should serialize at some point during its execution.* The RSS TM transition system makes a *guess* of when every transaction serializes. Depending upon the guess, each transaction has to follow certain restrictions on executable commands, if the transaction has to successfully commit.

Formally, we define an *RSS TM algorithm* A_{ss} as a family of RSS TM transition systems. The *RSS TM transition system* $A_{ss}^{n,k}$ for n threads and k variables is given by the tuple $\langle Q, q_{init}, D, \delta \rangle$. The thread state q^t is a 6-tuple $\langle Status^t, rs^t, ws^t, prs^t, pws^t, Preds^t \rangle$, where $Status^t \in \{\mathsf{started}, \mathsf{invalid}, \mathsf{serialized}, \mathsf{finished}\}$ is the status function, $rs^t \subseteq V$ is the read set, $ws^t \subseteq V$ is the write set, $prs^t \subseteq V$ is the prohibited read set, $pws^t \subseteq V$ is the prohibited write set, and $Preds^t \subseteq T$ is the predecessor set for thread t. If $v \in prs^t$ ($v \in pws^t$), then the status of the thread t is set to $\mathsf{invalid}$ if t globally reads (writes to) v. The initial thread state q_{init}^t is $\langle \mathsf{finished}, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset \rangle$. The set of extended commands is $D = C \cup \{\mathsf{serialize}\}$. The transition relation δ is non deterministic. A transition $(q_1, c, (d, t), r, q_2) \in \delta$ if c is enabled in q_1 for thread t and one of the following holds.

1. Local read. (i) $c = (\mathsf{read}, v)$ and $d = c$ and $r = 1$, and (ii) $v \in ws_1^t$, and (iii) $q_2 = q_1$. (When a thread reads v such that the read is not global, the state remains unchanged.)

2. Global read. (i) $c = (\mathsf{read}, v)$ and $d = c$ and $r = 1$, and (ii) $v \notin ws_1^t$, and (iii) if $status_1^t = \mathsf{finished}$ then $status_2^t = \mathsf{started}$, else if $status_1^t = \mathsf{serialized}$ and $v \in prs_1^t$, then $status_2^t = \mathsf{invalid}$, else $status_2^t = status_1^t$, and (iv) $rs_2^t = rs_1^t \cup \{v\}$ and $ws_2^t = ws_1^t$ and $prs_2^t = prs_1^t$ and $pws_2^t = pws_1^t$ and $Preds_2^t = Preds_1^t$, and (v) for all threads $u \neq t$, we have $q_2^u = q_1^u$. (When a thread t reads v globally, v is added to the read set. If the status of t is $\mathsf{finished}$, change the status of t to $\mathsf{started}$, else if the status of t is $\mathsf{serialized}$ and v is in the prohibited read set, then change status of t to $\mathsf{invalid}$.)

3. Write. (i) $c = (\mathsf{write}, v)$ and $d = c$ and $r = 1$, and (ii) if $status_1^t = \mathsf{finished}$ then $status_2^t = \mathsf{started}$, else if $status_1^t = \mathsf{serialized}$ and $v \in pws_1^t$, then $status_2^t = \mathsf{invalid}$, else $status_2^t = status_1^t$, and (iii) $ws_2^t = ws_1^t \cup \{v\}$ and $rs_2^t = rs_1^t$ and $prs_2^t = prs_1^t$ and $pws_2^t = pws_1^t$ and $Preds_2^t = Preds_1^t$, and (iv) for all threads $u \neq t$, we have $q_2^u = q_1^u$ (When a thread writes to v, the variable v is added to the write set. If the status of t is $\mathsf{finished}$, change the status to $\mathsf{started}$, else if the status is $\mathsf{serialized}$ and v is in the prohibited write set, then change status of t to $\mathsf{invalid}$.)

4. Serialize. (i) $d = \mathsf{serialize}$ and $r = \perp$, and (ii) $status_1^t = \mathsf{started}$, and (iii) $status_2^t = \mathsf{serialized}$ and $rs_2^t = rs_1^t$ and $ws_2^t = ws_1^t$ and $prs_2^t = prs_1^t$ and $pws_2^t = pws_1^t$ and $Preds_2^t = \{u \in T \mid Status_1^u = \mathsf{serialized}\}$, and (iv) for all threads $u \neq t$, we have $q_2^u = q_1^u$. (A thread t can serialize if the current status of t is $\mathsf{started}$, and the status of t is set to $\mathsf{serialized}$. Every thread whose status is $\mathsf{serialized}$ is added into the predecessor set of t.)

5. Commit. (i) $c = \mathsf{commit}$ and $d = c$ and $r = 1$, and (ii) $status_1^t \in \{\mathsf{serialized}, \mathsf{finished}\}$, and (iii) $status_2^t = \mathsf{finished}$ and $rs_2^t = ws_2^t = prs_2^t = pws_2^t = Preds_2^t = \emptyset$, and (iv) for all threads $u \neq t$, we have $rs_2^u = rs_1^u$ and $ws_2^u = ws_1^u$ and $Preds_2^u = Preds_1^u$, and (v) for all threads $u \neq t$, if $u \in Preds_1^t$, then $prs_2^u = prs_1^u \cup ws_1^t$ and $pws_2^u = pws_1^u \cup rs_1^t \cup ws_1^t$, otherwise $prs_2^u = prs_1^u$ and $pws_2^u = pws_1^u$, and (vi) for all threads $u \in Preds_1^t$, set $status_2^u = \mathsf{invalid}$ if $ws_1^u \cap ws_1^t \neq \emptyset$ or $ws_1^u \cap rs_1^t \neq \emptyset$, and $status_2^u = status_1^u$ otherwise (vii) for all threads $u \notin Preds_1^t$, set $status_2^u = \mathsf{invalid}$ if $ws_1^t \cap rs_1^u \neq \emptyset$, and $status_2^u = status_1^u$ otherwise (When a thread t commits, the current status of t should be $\mathsf{serialized}$ or $\mathsf{finished}$. The status of t is set to $\mathsf{finished}$. For every predecessor thread u of t, all variables in the write set of t are added to the prohibited read and write set of u. All variables in the read set of t are added to the prohibited write set of u. For all predecessor threads u of t, if the write set of u intersects with the read set or write set of t, the status of u is set to $\mathsf{invalid}$. For all threads u that are not predecessors of t such that the read set of u intersects with the write set of t, the status of u is set to $\mathsf{invalid}$.

For every state $q_1 \in Q$, a transition $(q_1, c, (\mathsf{abort}, t), 0, q_2)$ is in δ if $c \in C$ enabled in q_1 for thread t, and $rs_2^t = ws_2^t = prs_2^t = pws_2^t = Preds_2^t = \emptyset$, and $status_2^t = \mathsf{finished}$, and $q_2^u = q_1^u$ for all threads $u \neq t$.

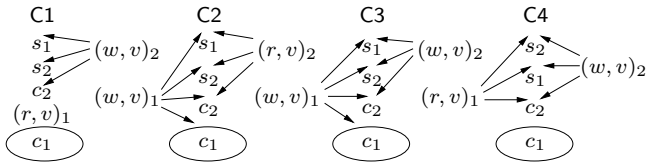


Figure 2. We use the same notation as in Table 1. The commits inside ovals are disallowed by the reference strictly serializable implementation. Each condition shows various cases. The arrows represent different possible positions for a command to occur in a given condition

Note that the non determinism in the transition relation comes from the $\mathsf{serialize}$ command, and the fact that abort is allowed in every state. For a reference TM transition system $A^{n,k}$, we define a *run* as a sequence $r = s_0 s_1 \dots s_n$ in $\hat{S}_D^*$ such that there exist states $q_0 \dots q_n$, commands $c_0 \dots c_n$, and responses $r_0 \dots r_n$ where (i) $q_0 = q_{init}$ and (ii) for all $j \geq 0$, we have $(q_j, c_j, s_j, r_j, q_{j+1}) \in \delta$. We define the language $L(A^{n,k})$ as the set of words w such that $w = r|_{\hat{S}}$ for some run r of $A^{n,k}$.

Theorem 2. Given a word w on n threads and k variables, the word w is strictly serializable if and only if $w \in L(A_{ss}^{n,k})$.

Proof. Consider an arbitrary run $r = s_0 s_1 \dots s_n$ of $A_{ss}^{n,k}$. Let $w = r|_{\hat{S}}$. Let w' be the sequential word such that w' is transaction equivalent to w and $x <_{w'} y$ if x serializes before y in the run r. Then, $com(w')$ is strictly equivalent to $com(w)$ iff for every transaction $x \in X$, the transaction x does not commit in r if one of the following conditions holds: (graphically shown in Figure 2):

C1. there exists a transaction y such that x serializes before y and y writes to a variable v and commits, and then x globally reads v

C2. there exists a transaction y such that x serializes before y and x writes to v and y reads v before x commits, and y commits

C3. there exists a transaction y such that x serializes before y and both x and y write to a variable v, and y commits before x does.

C4. there exists a transaction y such that x serializes after y and y writes to v and x reads v before y commits, and then y commits

The RSS TM transition system $A_{ss}^{n,k}$ guarantees by construction, that a transaction x does not commit in r if one of the conditions, C1-C4 holds. Hence, for every run r of $A_{ss}^{n,k}$, the word $w = r|_{\hat{S}}$ is strictly serializable.

Conversely, consider a word $w \in S^*$ on n threads and k variables such that w is strictly serializable. Thus, there is a sequential word w' such that $com(w')$ is strictly equivalent to $com(w)$). Let the committing transactions in the sequential word w' be given by the sequence $x_1 \dots x_k$ of transactions. Consider a run r of the RSS TM transition system $A_{ss}^{n,k}$ such that $w = r|_{\hat{S}}$ and for all i and j such that $i < j$, the transaction x_i serializes before x_j in r. The run r exists because (i) the RSS TM transition system guesses every possible serialization for every transaction during its execution, and (ii) given that w is strictly serializable, there is no transaction x in the sequence $x_1 \dots x_k$ that satisfies any of the conditions C1-C4, and commits in r. Thus, the word $w \in L(A_{ss}^{n,k})$. □

5.2 The reference TM algorithm for abort consistency

Apart from the requirements of the above mentioned reference TM algorithm for strict serializability, abort consistency requires that even global reads of aborting transactions observe consistent values. It turns out that we can even obtain a finite state representation of the RAC TM transition system by slightly modifying our RSS TM transition system.

The RAC TM transition system is based on the following notion: *Every transaction (committing, aborting, or pending) should serialize at some point during its execution*. Like the RSS TM transition system , the RAC TM transition system makes a *guess* of when every transaction serializes. Here, in addition to the RSS TM transition system, every transaction has to follow certain restrictions on executable commands, even to read some variable globally.

The formalism for *RAC TM algorithm* A_{ac} and the *RAC TM transition system* $A_{ac}^{n,k}$ is exactly similar to that of the RSS TM algorithm. The only difference comes in the transition relation δ, on a global read, and on a serialize command. We obtain the transition relation for $A_{ac}^{n,k}$ by replacing rules 2 and 4 of that of $A_{ss}^{n,k}$ by the rules 2a and 4a below. We only provide an informal description here for sake of brevity.

2a. Global read. When thread t reads v globally, v should not be in the prohibited read set. v is added to the read set. If the status of t is finished, it is changed to started. For every other thread u with status serialized such that t is not a predecessor of u, we add v to the prohibited write set of u, and we set status of u to invalid if v is in the write set of u.

4a. Serialize. When a thread t serializes, the current status of t should be started. The status of t is set to invalid if there is a thread u with status started and the read set of u intersects with the write set of t, otherwise, the status of t is set to serialized. All variables in read sets of threads with status started are added to the prohibited write set of t. All threads with status serialized are added to the predecessor set of t. For every other thread u, if the status of u is serialized and the write set of u intersects with the read set of t, then the status of u is set to invalid. For every thread u with status serialized, the read set of t is added to the prohibited write set of u.

Theorem 3. Given a word w on n threads and k variables, the word w is abort consistent if and only if $w \in L(A_{ac}^{n,k})$.

5.3 Implementation and simulation checking

A TM M defined by a TM algorithm A ensures strict serializability if $L(A^{2,2}) \subseteq L(A_{ss}^{2,2})$. As checking language inclusion is PSPACE-hard, we use the common technique of checking for the existence of a simulation relation between both transition systems. The existence of a simulation relation is a sufficient condition for language inclusion. We write $A_1^{2,2} \prec A_2^{2,2}$ to denote that there exists a simulation relation between $A_1^{2,2}$ and $A_2^{2,2}$. For a TM M defined by a TM algorithm A which satisfies the symmetry assumptions of the reduction theorem (Theorem 1), the following hold: (i) The TM M ensures strict serializability (resp. abort consistency) if $A^{2,2} \prec A_{ss}^{2,2}$ (resp. $A^{2,2} \prec A_{ac}^{2,2}$). (ii) M does not ensure strict serializability (resp. abort consistency) if there exists a word $w \in \hat{S}^*$ such that $w \in L(A^{2,2})$ and $w \notin L(A_{ss}^{2,2})$ (resp. $w \notin L(A_{ac}^{2,2})$).

We built an automatic verification tool in C for checking the existence of simulation relations using the quadratic algorithm by Henzinger et al. [HHK95]. The tool is conceived as a platform for the automatic verification of TMs that satisfies the symmetry properties. We mention that simulation checking requires extra technical care in this scenario due to different extended alphabet in different TMs. The tool takes as input two TM algorithms A_1 and A_2, and checks whether $A_1^{2,2} \prec A_2^{2,2}$. If the tool fails to find a simulation relation, it attempts to return a counterexample $w \in \hat{S}^*$ such that $w \in L(A^{2,2})$ and $w \notin L(A_{ss}^{2,2})$. However, in certain cases, it is possible that even though language inclusion holds, the tool cannot find a simulation relation. Thus, our decision procedure is sound but not complete. It turns out that for the TM transition systems that we considered, our tool terminates after proving the simulation relation, or after finding a counterexample.

Table 2. Time for simulation checking for TM algorithms on a quad dual core 2.8 GHz server with 16 GB RAM. In case simulation holds, we write YES followed by the time required for the simulation. Otherwise, we write NO followed by the counterexample produced, followed by the time required to prove that no simulation exists and to find the counterexample.

TM transition system $A^{2,2}$	Number of states	$A^{2,2} \prec A_{ss}^{2,2}$	$A^{2,2} \prec A_{ac}^{2,2}$
seq	3	YES, $0.8s$	YES, $0.7s$
$2PL$	99	YES, $13s$	YES, $8s$
$dstm$	944	YES, $127s$	YES, $82s$
$TL2$	4160	YES, $583s$	YES, $387s$
occ	4480	YES, $765s$	NO, w_1, $569s$
$TL2$ modified	5480	NO, w_2, $887s$	NO, w_2, $674s$
ss	12346	—	—
ac	9202	—	—
Counterexample			
w_1	$(w,1)_2, (r,1)_1, c_2, (r,1)_1$		
w_2	$(w,2)_2, (r,2)_1, (w,2)_1, c_2, c_1$		

The results of our simulation checks are presented in Table 2. Our results demonstrate that all TMs discussed in Section 3 — sequential, 2PL, DSTM, and TL2— are simulated by both reference TM transition systems. As for the OCC TM, it is simulated by the RSS TM transition system, but not by the RAC TM transition system. The tool gives a counterexample in the latter case.

Theorem 4. The sequential TM, two phase locking TM, DSTM, and TL2 TM ensure abort consistency. The optimistic concurrency control TM ensures strict serializability, but not abort consistency.

We also experimented with a subtle point in the TL2 algorithm. We interchanged the order of the commands lock and validate in the TL2 to obtain modified TL2 TM algorithm. We first do validate, then lock the variables, and then perform chklock. The tool found that the modified TL2 is not simulated by either of the reference TM transition systems, and provided counterexamples corresponding to both simulation checks. Thus, we conclude that the modified TL2 algorithm does not ensure abort consistency, or even the weaker safety criterion of strict serializability.

5.4 Comparing TM algorithms

In our framework, we can also compare the languages of different TM transition systems. Checking language inclusion between TM transition systems provides information about liberality of different TM implementations, i.e., which TM algorithm has strictly more words than another. Liberality can be one of the important criteria for ranking different TM algorithms.

We compare the sequential, 2PL, DSTM, and TL2 TMs for liberality. For this purpose, we need to define an additional symmetry property, P5, which is satisfied by these TMs. For TMs that ensure abort consistency, and satisfy the properties P1–P5, we can show the reduction theorem that, if $L(A_1^{2,2}) \subseteq L(A_2^{2,2})$, then $L(A_1^{n,k}) \subseteq L(A_2^{n,k})$ for arbitrary n and k.

P5. For every word w such that there is no program p where w is a finite prefix of a word in $M(p)$, one of the following holds: (i) w is not abort consistent, or (ii) there exists a word w' such that for no program p', the word $w' \in M(p')$, where w' is obtained as follows. All aborting transactions of w are removed, then a transaction projection is taken on transactions of any two threads, then a variable projection is taken on any two variables to obtain word w'. This property just states that when an abort consistent word w is not produced by a TM, then it is due to *local conflicts* on two threads and two variables. This is due to the fact

Table 3. Ranking different transactional memories. The time is measured on a 2.66 GHz dual core desktop PC with 2 GB of RAM. The notation is similar to that in Table 2

$A^{2,2}$	$\prec A^{2,2}_{seq}$	$\prec A^{2,2}_{2PL}$	$\prec A^{2,2}_{dstm}$	$\prec A^{2,2}_{TL2}$
seq	—	YES, $0.1s$	YES, $0.2s$	YES, $0.4s$
$2PL$	NO, w_1, $0.3s$	—	YES, $0.6s$	YES, $2.1s$
$dstm$	NO, w_1, $0.7s$	NO, w_2, $2.4s$	—	YES, $13s$
$TL2$	NO, w_1, $0.8s$	NO, w_2, $4s$	NO, w_3, $17s$	—

Counterexample	
w_1	$(r,2)_2, c_1$
w_2	$(w,1)_2, (r,1)_1$
w_3	$(w,2)_1, (w,2)_2, c_1$

that conventional TM algorithms use techniques like validating the read set, and locking the write set, to guarantee correctness.

Formally, a TM M_1 defined by a TM algorithm A_1 is *more liberal* than a TM M_2 defined by a TM algorithm A_2 (denoted as $M_1 \geq M_2$) if $L(A_2^{2,2}) \subseteq L(A_1^{2,2})$. As in the previous subsection, we check language inclusion by checking the existence of a simulation relation. Our results are listed in Table 3. The following theorem follows.

Theorem 5. TL2-TM $\geq$ DSTM $\geq$ 2PL-TM $\geq$ sequential-TM.

6. Verifying liveness

We define two different notions of liveness, obstruction freedom and livelock freedom, as discussed in the TM literature. The third notion, wait freedom, implies livelock freedom. Since we will show that none of our example TMs satisfy livelock freedom, they do not satisfy wait freedom either.

Obstruction freedom [HLM03] requires that if a thread performs an infinite number of commands in isolation, where the commands include an infinite number of aborts, then the commands include an infinite number of commits. An infinite word $w \in \hat{S}^\omega$ is *obstruction free* if $\bigwedge_{t\in T}(\Box\Diamond((\mathsf{commit},t)\vee(c,u))\vee\Diamond\Box\neg(\mathsf{abort},t))$, where $c \in \hat{C}$ and $u \neq t$. This is a Streett condition.

Livelock freedom [AKH03] requires that on every infinite trace, an infinite number of commits are executed. An infinite word $w \in \hat{S}^\omega$ is *livelock free* if $\Box\Diamond(\bigvee_{t\in T}(\mathsf{commit},t))\vee\Diamond\Box(\bigwedge_{t\in T}\neg(\mathsf{abort},t))$. This implies obstruction freedom.

A TM M *ensures obstruction freedom (resp. livelock freedom) for all programs with n threads and k variables* if for every program $p \in P^{n,k}$, every word $w \in M(p)$ is obstruction free (resp. livelock free). A TM M *ensures obstruction freedom (resp. livelock freedom)* if M ensures obstruction freedom (resp. livelock freedom) for all programs with arbitrary number of threads and variables. A TM M ensures obstruction freedom if it ensures livelock freedom. We use the formalism of TM algorithms for verifying of liveness properties in TM. We define a *loop* l in a TM transition system $A^{n,k}$ as a word $s_0 \dots s_m$ such that there exist a set of states $q_0 \dots q_m$ in $A^{n,k}$ such that for all i where $0 \leq i < m$, we have $(q_i, c_i, s_i, r_i q_{i+1}) \in \delta$ and $(q_m, c_m, s_m, r_m, q_0) \in \delta$, where δ is the transition relation of $A^{n,k}$.

Although obstruction freedom is formally a Streett condition, the different conjuncts (Streett pairs) do not overlap, which permits a simple model checking procedure. In particular, a TM M defined by a TM algorithm A ensures obstruction freedom for all programs with n threads and k variables iff there does not exist a loop l in $A^{n,k}$ such that all commands in l are from the same thread, and l has no commit, and l has an abort. Similarly, since livelock freedom is a single-pair Streett condition, a TM M ensures livelock freedom for all programs with n threads and k variables iff there does not exist a loop l in $A^{n,k}$ such that there is no commit in l, and every thread that has a command in l aborts in l.

6.1 Reduction theorem for liveness

As we did for safety, we state a reduction theorem that proves that it is sufficient to verify liveness of a TM on programs with two threads and two variables to generalize the result to all programs. For this purpose, we need two more symmetry properties of TM algorithms. These properties are again satisfied by all TM algorithms that we have discussed. Let $w = w_1 \cdot w_2$ be an infinite word such that w is in $M(p)$ for some program p, and no pending transaction in w_1 has a statement in w_2, and all the commands in w_2 are from the same thread. For $i \in \{1,2\}$, let V_i be the variables accessed in w_i.

P6. *Transaction projection.* Let w_1' be the word obtained by taking the transaction projection of w_1 on non aborting transactions. Then $w_1' \cdot w_2 \in M(p')$ for some program p'. Moreover, if w_1 has no aborting transactions, there exists a word $w' = w_1'' \cdot w_2 \in M(p)$, where w_1'' is obtained by projecting w_1 to transactions of thread t, where t has commands in w_1.

P7. *Variable projection.* There exists a word $w' = w_1 \cdot w_2'$ such that w_2' is the variable projection of w_2 on $\{v\}$, where $v \in V_2$, and w' is in $M(p')$ for some program p'. Moreover, if w_1 has no aborting transactions, then the word $w' = w_1' \cdot w_2$ is in $M(p')$ for some program p', where w_1' is the variable projection of w_1 on V_2.

Theorem 6. If a TM M satisfies properties P6 and P7, then M ensures obstruction freedom if M ensures obstruction freedom for two threads and one variable.

Proof. Given a $w \in M(p)$ on arbitrary number of threads and variables such that w is not obstruction free, we can use properties P6 and P7 to obtain a word w' on two threads and one variable such that $w' \in M(p')$ for some program p'. □

6.2 Model checking liveness

We extended our verification tool to check obstruction freedom and livelock freedom properties for transaction memories defined by TM algorithms A. To check obstruction freedom, our tool tries to find a loop l in $A^{2,1}$ such that all commands in w are from the same thread, and w has no commit, and w has an abort. If the tool finds such a loop, the loop is a counterexample to obstruction freedom. If the tool does not find a loop, we know that the TM ensures obstruction freedom. In this way, our tool provides a platform for TM designers to check which liveness properties are ensured by their TMs. If the liveness property fails, then the tool provides feedback in the form of an execution trace that represents a counterexample.

Our results are shown in Table 4. The next theorem follows.

Theorem 7. DSTM ensures obstruction freedom and does not ensure livelock freedom. Sequential TM, 2PL TM, TL2 TM, and optimistic concurrency control TM do not ensure obstruction freedom.

7. Related Work

Very recently, the article [COP+07] has been brought to our attention. While their goals are similar to ours, as far as we can tell, they check only the correctness of finite instances (e.g., STMs applied to programs with a small number of threads and variables), without offering reduction theorems that establish the sufficiency of such checks. Moreover, they consider only the strong safety criteria of [Sco06], which fail, for example, for TL2. Also, the article [COP+07] does not address the verification of liveness properties.

Our construction of the reference STM algorithms is related to the work of Fle and Roucairol [FR85]. They investigated the set of concurrent traces that are generated by a finite set of iterating

Table 4. Results of model checking liveness on a dual core 2.66GHz desktop PC with 2 GB RAM. The notation is similar to Table 2. The counterexamples obtained are of the form $a \cdot b^{\omega}$. We write the looping part b here.

TM algorithm	Obstruction freedom	Livelock freedom
seq	NO, w_1, $0.1s$	NO, w_1, $0.1s$
$2PL$	NO, w_1, $0.1s$	NO, w_1, $0.1s$
$dstm$	YES, $2s$	NO, w_2, $0.2s$
$TL2$	NO, w_1, $0.4s$	NO, w_1, $0.4s$
occ	NO, w_3, $0.7s$	NO, w_3, $0.7s$

Counterexamples	
w_1	a_1
w_2	$a_1, (r,1)_1, (o,1)_1, a_2, (o,1)_2$
w_3	s_1, a_1

transactions. They proved that the language consisting of all traces that are conflict equivalent to a sequential trace is regular. However, their results cannot be applied in the presence of aborting transactions, as they require transitivity of conflicts, which does not hold when transactions may abort.

There has been much research in the verification of relaxed memory models and cache-coherence protocols for modern multiprocessors, e.g., [HQR99]. In most of this work, the semantics of a shared memory is given by a *memory consistency model*, which defines the possible outcomes of executing concurrent programs. For example, in order to determine if a processor complies with its memory model, Gopalakrishnan et. al [GYS04] provided a method to establish if a given back-annotated execution trace of a processor is valid with respect to its memory model. The specification of the processor's memory model is translated into a HOL specification and a QBF-solver is used to establish the corresponding memory ordering constraints taking the given execution trace into account. Burckhardt et. al [BAM07] developed a method based on SAT-based bounded model checking to verify concurrent data types on different memory models by testing exhaustively all concurrent executions of a given test program. In comparison, our work is more general as it targets an STM without a particular program in mind. On the other hand, since it specifically targets STM and, correspondingly, uses a deferred update semantics rather than a memory consistency model, our approach is also more restrictive.

8. Conclusion

We presented a new technique for verifying STM safety and liveness properties. The cornerstones of our technique are a finite-state representation for the languages of strictly serializable and abort consistent executions, and an automated verification tool for STMs. Our method applies to all STM protocols that feature certain symmetry properties, and we successfully verified abort consistency for 2PL, DSTM, and TL2, and the obstruction freedom of DSTM.

Although most STM protocols we know of fulfill the required symmetry properties, some do not, and these open interesting research opportunities. In particular, our symmetry properties do not hold in cases when aborting transactions are given priority (in general when history matters in making decisions). Similarly, our framework does not apply when transactions help each other. For instance, we cannot model for example Fraser's STM [FH07] nor the *Karma* contention manager of Scherer and Scott [SS05]. Also, our liveness properties capture deterministic notions. It would be interesting to account for probabilistic means to deal with contention, such as random exponential backoff.

We also assumed that the commands in the extended alphabet we considered are executed atomically. So, STM implementations have to guarantee this level of atomicity to ensure correctness using our methodology. It is not clear how to reason about correctness if the lower-level primitives are not atomic. It is also an interesting open question to compare the liberality of STMs when extended commands are not atomic.

References

[AKH03] James H. Anderson, Yong-Jik Kim, and Ted Herman. Shared-memory mutual exclusion: major research trends since 1986. *Distributed Computing*, pages 75–110, 2003.

[AMP00] Rajeev Alur, Kenneth L. McMillan, and Doron Peled. Model-checking of correctness conditions for concurrent objects. *Information and Computation*, pages 167–188, 2000.

[BAM07] Sebastian Burckhardt, Rajeev Alur, and Milo M. K. Martin. Checkfence: checking consistency of concurrent data types on relaxed memory models. In *PLDI*, pages 12–21, 2007.

[BCG89] M. C. Browne, E. M. Clarke, and O. Grumberg. Reasoning about networks with many identical finite state processes. *Inf. Comput.*, pages 13–31, 1989.

[COP+07] Ariel Cohen, John O'Leary, Amir Pnueli, Mark R. Tuttle, and Lenore Zuck. Verifying correctness of transactional memories. In *FMCAD*, 2007.

[DSS06] David Dice, Ori Shalev, and Nir Shavit. Transactional locking ii. In *DISC*, pages 194–208, 2006.

[EGLT76] Kapali P. Eswaran, Jim Gray, Raymond A. Lorie, and Irving L. Traiger. The notions of consistency and predicate locks in a database system. *Commun. ACM*, pages 624–633, 1976.

[FH07] Keir Fraser and Tim Harris. Concurrent programming without locks. *ACM Trans. Comput. Syst.*, 2007.

[FR85] Marie-Paule Flé and Gérard Roucairol. Maximal serializability of iterated transactions. *TCS*, pages 1–16, 1985.

[GK08] Rachid Guerraoui and Michał Kapałka. On the correctness of transactional memory. In *PPoPP*, 2008. to appear.

[GYS04] Ganesh Gopalakrishnan, Yue Yang, and Hemanthkumar Sivaraj. QB or Not QB: An efficient execution verification tool for memory orderings. In *CAV*, pages 401–413, 2004.

[Her91] Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, pages 124–149, 1991.

[HHK95] Monika Rauch Henzinger, Thomas A. Henzinger, and Peter W. Kopke. Computing simulations on finite and infinite graphs. In *FOCS*, pages 453–462, 1995.

[HLM03] Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free synchronization: Double-ended queues as an example. *ICDCS*, page 522, 2003.

[HLMS03] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer. Software transactional memory for dynamic-sized data structures. In *PODC*, pages 92–101, 2003.

[HM93] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA*, pages 289–300, 1993.

[HQR99] Thomas A. Henzinger, Shaz Qadeer, and Sriram K. Rajamani. Verifying sequential consistency on shared-memory multiprocessor systems. In *CAV*, pages 301–315, 1999.

[JD94] J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessors control. In *CAV*, pages 68–80, 1994.

[KR81] H. T. Kung and John T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, pages 213–226, 1981.

[LR07] James R. Larus and Ravi Rajwar. *Transactional Memory*. Synthesis Lectures on Computer Architecture. Morgan & Claypool, 2007.

[Pap79] Christos H. Papadimitriou. The serializability of concurrent

database updates. *J. ACM*, pages 631–653, 1979.

[Sco06] Michael L. Scott. Sequential specification of transactional memory semantics. In *ACM SIGPLAN WTC*, 2006.

[SS05] William N. Scherer and Michael L. Scott. Advanced contention management for dynamic software transactional memory. In *PODC*, pages 240–248, 2005.

[ST95] Nir Shavit and Dan Touitou. Software transactional memory. In *SPDC*, pages 204–213, 1995.