Model checking transactional memories

**Rachid Guerraoui · Thomas A. Henzinger ·
Vasu Singh**

Abstract Model checking transactional memories (TMs) is difficult because of the unbounded number, length, and delay of concurrent transactions, as well as the unbounded size of the memory. We show that, under certain conditions satisfied by most TMs we know of, the model checking problem can be reduced to a finite-state problem, and we illustrate the use of the method by proving the correctness of several TMs, including two-phase locking, DSTM, and TL2. The safety properties we consider include strict serializability and opacity; 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 TMs mentioned above. In a first step we show that every TM that enjoys certain structural properties either violates a requirement on some program with two threads and two shared variables, or satisfies the requirement on all programs. In the second step, we use a model checker to prove the requirement for the TM applied to a most general program with two threads and two variables. In the safety case, the model checker checks language inclusion between two finite-state transition systems, a nondeterministic transition system representing the given TM applied to a most general program, and a deterministic transition system

representing a most liberal safe TM applied to the same program. The given TM transition system is nondeterministic because a TM can be used with different contention managers, which resolve conflicts differently. In the liveness case, the model checker analyzes fairness conditions on the given TM transition system.

Keywords Transactional memories · Model checking

1 Introduction

Transactional memory (TM) has recently gained much interest due to the advent of multicore architectures. A TM allows to structure an application in terms of coarse-grained code blocks that appear to be executed atomically [20,26]. A TM provides the illusion of sequentiality to a programmer and maximal flexibility to the underlying hardware. However, behind the apparent simplicity of the TM abstraction, lie challenging algorithms that seek to ensure transactional atomicity without restricting parallelism.

Inspired by how databases manage concurrency, TM was first introduced by Herlihy and Moss [20] in multi-processor design. Later Shavit and Touitou [26] introduced STM, a software-based variant of the concept. Despite the large amount of experimental work on TMs [21], little effort has been devoted to their formalization [15,25]. Two safety properties, strict serializability [22] and opacity [15], have been considered for TMs. The former requires committed transactions to appear as if executed at indivisible points in time during their lifetime. Opacity goes a step further and also requires aborted transactions to always access consistent state. The notion of opacity conveys an emerging consensus about correctness in the TM community [7,19]. The liveness requirements we consider are the standard notions of

R. Guerraoui
LPD, I&C, EPFL, Station 14, 1015 Lausanne, Switzerland
e-mail: rachid.guerraoui@epfl.ch

T. A. Henzinger · V. Singh (✉)
MTC, I&C, EPFL, Station 14, 1015 Lausanne, Switzerland
e-mail: vasu.singh@epfl.ch

T. A. Henzinger
e-mail: tah@epfl.ch

obstruction freedom [18], livelock freedom [2], and wait freedom [17]. Obstruction freedom requires that if a transaction executes in isolation, then it eventually commits. Livelock freedom requires some transaction to eventually commit. Wait freedom requires every transaction to eventually commit.

Precisely because TMs encapsulate the difficulty of handling concurrency, the potential of subtle errors in their implementation is enormous. This makes TM a ripe and important proving ground for formal verification. However, three major challenges need to be tackled to model check TMs.

1. Transactional memories, being highly performance oriented, employ sophisticated techniques to ensure correctness in the face of conflicts due to concurrency. Moreover, TMs generally rely on a separate module, called a contention manager, to resolve conflicts when they occur, and to guarantee liveness. A first step towards verification is to create a formalism to express different TMs and contention managers in a uniform framework.
2. 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 TM: 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. Special care is needed in formulating a verification problem that is both relevant and solvable, as some problems about sequentializing concurrent systems are undecidable [1].
3. The specification of a TM universally quantifies over all possible application programs, requiring the desired safety and liveness conditions *for all* programs that are executed on the TM.

We present in this paper a new technique for verifying TM safety and liveness properties. We first provide a framework to formalize TM together with specific contention managers as TM algorithms, as well as TM safety and liveness properties. Then, we exploit the structural symmetries that are inherent in TM algorithms to reduce the verification of unbounded TM state spaces to a problem that involves only a small number of threads and shared variables. Specifically, we show that every TM that enjoys certain structural properties either violates any of the considered safety and liveness requirements with two threads and two shared variables, or always satisfies the requirement. Basically, these structural properties expect all threads to be treated equally. They are fulfilled by most TMs, including for instance, two-phase locking, DSTM [19], and TL2 [7]. 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 [3] and cache-coherence protocols [16].

Finally, we define two finite-state deterministic transition systems, called *TM specifications* that generate exactly the strictly serializable (resp. opaque) executions of programs with two threads and two shared variables. These transition systems are obtained by applying the most liberal TM safe with respect to strict serializability (resp. opacity) to a most general program. 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 two), a set of written variables (at most two), a set of variables not allowed to be read (at most two), a set of variables not allowed to be written (at most two), a set of threads with commit-dependent predecessor transactions (at most one), and a set of independent predecessor transactions (at most one). As it is difficult to directly prove the correctness of the deterministic TM specifications, we provide more natural nondeterministic TM specifications for strict serializability and opacity, and prove their correctness. Then, we use an antichain-based tool [28] and show automatically that these nondeterministic TM specifications are language equivalent to their deterministic counterparts.

Putting all steps together, we reduce the problem of verifying the safety of transactional memories, which is unbounded in many dimensions (memory size, thread count, transaction delay, etc.), to a language inclusion check between a nondeterministic and a deterministic finite-state system. Since the TM specification is deterministic, language inclusion can be checked in time linear in the size of the systems: For two-phase locking, DSTM [19], and TL2 [7], we obtain transition systems with up to 20,000 states for the most general program with 2 threads and 2 variables. We implemented a checker that automatically verifies strict serializability and opacity for two-phase locking, DSTM, and TL2 in less than a minute. The liveness properties guaranteed by a TM depend on the specific contention manager used with the TM. Generally a TM, by itself, does not guarantee any interesting liveness properties. So, for liveness, we model check a TM together with a specific contention manager to determine which liveness property is satisfied. We again prove a structural reduction theorem to check the desired liveness requirement on the finite-state transition system that results from a given TM algorithm applied to a most general program with two threads and one variable. Our tool checked the different liveness properties. In the case of obstruction freedom, this amounts to checking a Streett condition [27]. For instance, the check goes through for DSTM with the aggressive contention manager. For two-phase locking and TL2 with the polite contention manager, the model checker automatically generates counterexamples to obstruction freedom.

Our methodology is applicable to any TM algorithm that satisfies the structural properties. We find that correctness

is not self-evident in many TM algorithms. For example, we found an ambiguity in ordering of two particular operations in the published TL2 algorithm [7]. One of the orderings makes TL2 unsafe. In this case, the check for language inclusion provides as counterexample an execution that is not strictly serializable (and thus not opaque). We therefore expect our verification tool to be useful to TM designers when they develop or modify TM algorithms.

2 Framework

We present a framework to express transactions and their correctness properties.

2.1 Preliminaries

Let V be a set $\{1, \ldots, k\}$ of k *variables*, and let $C = \{\mathsf{commit}\} \cup (\{\mathsf{read}, \mathsf{write}\} \times V)$ be the set of *commands* on the variables V. Also, let $\hat{C} = C \cup \{\mathsf{abort}\}$. Let $T = \{1, \ldots, n\}$ be a set of n *threads*. Let $\hat{S} = \hat{C} \times T$ be the set of *statements*. Also, let $S = C \times T$. A *word* $w \in \hat{S}^*$ is a finite sequence of statements. Given a word $w \in \hat{S}^*$, we define the *thread projection* $w|_t$ of w on thread $t \in T$ as the sub-sequence of w consisting of all statements s in w such that $s \in \hat{C} \times \{t\}$. Given a thread projection $w|_t = s_0 \ldots s_m$ of a word w on thread t, a statement s_i is *finishing in* $w|_t$ if it is a commit or an abort. 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.

2.2 Transactions

Given a thread 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 (i) s_0 is initiating in $w|_t$, and (ii) s_m is either finishing in $w|_t$, or s_m is the last statement in $w|_t$, and (iii) no other statement in x is finishing in $w|_t$. The transaction x is *committing* in w if s_m is a commit. The transaction x is *aborting* in w if s_m is an abort. Otherwise, the transaction x is *unfinished* 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 last statement of x occurs before the first 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 subsequence of w which consists of every statement in w that is a part of a committing transaction. 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 transac-

tion 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.

2.3 Safety properties of TM

We consider two safety properties for transactional memories: strict serializability and opacity. Intuitively, strict serializability [22] requires that the order of conflicting statements from committing transactions is preserved, and the order of non-overlapping transactions is preserved. Opacity [15], in addition to strict serializability, requires that even aborting transactions do not read inconsistent values. The motivation behind the stricter requirement for aborting transactions in opacity is that in TMs, inconsistent reads may have unexpected side effects, like infinite loops, or array bound violations.

We define that a statement s_1 of transaction x and a statement s_2 of transaction y (where x is different from y) *conflict* in a word w if (i) s_1 is a global read of some 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 some variable v. This notion of conflict corresponds to the deferred update semantics [21] in transactional memories, where the writes of a transaction are made global upon the commit. Our methodology can be adapted for direct update semantics by changing the definition of a conflict.

A word $w = s_0 \ldots s_m$ is *strictly equivalent* to a word w' if (i) for every thread $t \in T$, we have $w|_t = w'|_t$, 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', and (iii) for every pair x, y of transactions in w, where x is a committing or an aborting transaction, if $x <_w y$, then it is not the case that $y <_{w'} x$.

We define the safety property *strict serializability* $\pi_{ss} \subseteq \hat{S}^*$ as the set of words w such that there exists a sequential word w', where w' is strictly equivalent to $com(w)$. Furthermore, we define *opacity* $\pi_{op} \subseteq \hat{S}^*$ as the set of words w such that there exists a sequential word w', where w' is strictly equivalent to w. We note that $\pi_{op} \subseteq \pi_{ss}$, that is, if a word is opaque, then it is strictly serializable.

2.4 Liveness properties of TM

We define two different notions of liveness, obstruction freedom and livelock freedom, as discussed in the TM literature. A third notion, wait freedom [17], implies livelock freedom. Since we will show that none of our TM examples satisfy livelock freedom, they do not satisfy wait freedom either.

We consider infinite words on $\hat{S}^\omega$. An infinite word $w \in \hat{S}^\omega$ is *obstruction free* [18] if for all threads t, if the word w has an infinite number of aborts of t, then w has an infinite number of commits of t, or there are infinitely many statements of some thread $u \neq t$. Formally, w is *obstruction free*

if $\bigwedge_{t \in T}(\square\lozenge(\mathsf{abort}, t) \to \square\lozenge((\mathsf{commit}, t) \vee \bigvee_{c \in \hat{C}, u \in T \setminus \{t\}} (c, u)))$, where the temporal operation $\square$ denotes 'always' and the temporal operation $\lozenge$ denotes 'eventually'. Obstruction freedom is a Streett condition [27].

An infinite word $w \in \hat{S}^\omega$ is *livelock free* [2] if the word has an infinite number of commits, or there is a thread t such that t has infinitely many statements and finitely many aborts in w. Formally, w is *livelock free* if $\square\lozenge(\bigvee_{t \in T}(\mathsf{commit}, t)) \vee \bigvee_{t \in T}(\square\lozenge(\bigvee_{c \in C}(c, t)) \wedge \lozenge\square\neg(\mathsf{abort}, t))$. Note that livelock freedom implies obstruction freedom. This is because if a word w has an infinite number of commits, or if w has infinitely many statements and finitely many aborts in w, then w is obstruction free.

2.5 TM specifications for safety

We capture safety properties of TM using TM specifications. A *TM specification* is a 3-tuple $\langle Q, q_{init}, \delta \rangle$, where Q is a set of states, q_{init} is the initial state, and $\delta \subseteq Q \times ((\hat{C} \cup \{\varepsilon\}) \times T) \times Q$ is a transition relation. A finite word $s_0 \ldots s_m$ in $\hat{S}^*$ is a *run* of the TM specification if there exist states $q_0 \ldots q_{m+1}$ in Q such that $q_0 = q_{init}$, and for all i such that $0 \le i \le m$, we have either $(q_i, s_i, q_{i+1}) \in \delta$, or $(q_i, (\varepsilon, t), q_{i+1}) \in \delta$. The *language L* of a TM specification is the set of all runs of the TM specification. A *TM specification* Σ defines a correctness property π if $L(\Sigma) = \pi$. A TM specification is *deterministic* if for every state $q \in Q$, we have (i) for every statement $s \in \hat{S}$, there is at most one state $q' \in Q$ such that $(q, s, q') \in \delta$, and (ii) there is no state $q' \in Q$ such that $(q, (\varepsilon, t), q') \in \delta$.

We shall provide both nondeterministic and deterministic TM specifications for strict serializability and opacity.

2.6 Transactional memories

We characterize a TM by the set of infinite words it can produce. Formally, a *transactional memory (TM) M* is a subset of $\hat{S}^\omega$. We say that *M ensures* (n, k) *strict serializability (resp. (n, k) opacity)* if for every prefix w of every word in M such that w has at most n threads and at most k variables, we have $w \in \pi_{ss}$ (resp. $w \in \pi_{op}$). Moreover, *M ensures strict serializability (resp. opacity)* if M ensures (n, k) strict serializability (resp. opacity) for all n and k. A TM *M ensures* (n, k) *obstruction freedom (resp. (n, k) livelock freedom)* if every word $w \in M$ such that w has at most n threads and at most k variables is obstruction free (resp. livelock free). Moreover, *M ensures obstruction freedom (resp. livelock freedom)* if M ensures (n, k) obstruction freedom (resp. (n, k) livelock freedom) for all n and k.

In practice, TMs may employ a separate module, called a *contention manager*, to enhance liveness [14,24]. A contention manager resolves conflicts on the basis of the past behavior of the transactions. Various contention managers have been proposed in the literature. For example, the *Karma* contention manager prioritizes transactions according to the number of objects opened, whereas the *Backoff* contention manager backs off conflicting transactions for a random duration [24]. When the transactional memory detects a conflict, it requests the contention manager to resolve the conflict. The contention manager proposes the TM the next statement to be executed. A TM M and a contention manager cm define a new transactional memory $M_{cm} \subseteq \hat{S}^\omega$.

3 TM Algorithms

We now present a formalism to express various TMs using TM algorithms. A TM algorithm consists of a set of states, an initial state, an extended set of commands depending on the underlying TM, a conflict function, a pending function, and a transition relation. A command is executed as a sequence of extended commands, all of which execute atomically. Thus, the extended commands include the set C of commands, as well as TM specific additional commands. For example, a given TM may require that a thread locks a variable before writing to the variable. The conflict function captures the statements in a state, when the TM algorithm may consult a contention manager for a decision. The pending function represents the pending command of a thread in a state, and ensures that if a thread has not finished the execution of all extended commands corresponding to a particular command, then no other command is executed by the thread.

We define a *TM algorithm* $A = \langle Q, q_{init}, D, \phi, \gamma, \delta \rangle$, where

- Q is a set of states,
- q_{init} is the initial state,
- $D \supseteq C$ is the set of extended commands,
- $\phi : Q \times S \to \mathbb{B}$ is the conflict function,
- $\gamma : Q \times T \to C \cup \{\bot\}$ is the pending function, and
- $\delta \subseteq Q \times C \times \hat{S}_D \times Resp \times Q$ is the transition relation, where $\hat{S}_D = (D \cup \{\mathsf{abort}\}) \times T$ and $Resp = \{\bot, 0, 1\}$ is the set of responses.

For a TM algorithm $A = \langle Q, q_{init}, D, \phi, \gamma, \delta \rangle$, the following rules hold:

- No command is pending in the initial state for all threads. For all threads $t \in T$, we have $\gamma(q_{init}, t) = \bot$.
- If there is an incoming transition to state q' of thread t with command c and response $\bot$, then c is pending in q' for t. For all states $q, q' \in Q$ such that there is an incoming transition $(q, c, (d, t), r, q')$ to q' in δ, if $r = \bot$, then $\gamma(q', t) = c$, otherwise $\gamma(q', t) = \bot$.
- On a transition of a thread, the pending command of other threads does not change. For all states $q, q' \in Q$

s.t. there is an incoming transition $(q, c, (d, t), r, q')$ to q' in δ, we have $\gamma(q', u) = \gamma(q, u)$ for all threads $u \neq t$.

- If a command is pending in a state for a thread t, then all outgoing transitions from the state by t are for the pending command. For all states q and all threads t, if $\gamma(q, t) = c$ with $c \neq \perp$, then for all outgoing transitions $(q, c_1, (d, t), r, q')$ from q in δ, we have $c_1 = c$.

- If no command is pending in a state for a thread t, then there is an outgoing transition from the state by thread t for every command. For all states q and all threads t, if $\gamma(q, t) = \perp$, then there is an outgoing transition $(q, c, (d, t), r, q')$ from q in δ for every command $c \in C$.

- For all transitions with the extended command as abort, the response is 0. For all $q \in Q$, for all transitions $(q, c, (d, t), r, q')$ in δ, we have $d =$ abort if and only if $r = 0$.

- For all states, there is at most one transition corresponding to a given command c, a given extended command d, and a given thread t. For all $q \in Q$, if $(q, c, (d, t), r_1, q') \in \delta$ and $(q, c, (d, t), r_2, q'') \in \delta$, then $r_1 = r_2$ and $q' = q''$.

- For all states, if a statement s does not conflict in the state, then there is at most one outgoing transition corresponding to s from the state. For all $q \in Q$ and $c \in C$, if $(q, c, (d_1, t), r_1, q') \in \delta$ and $(q, c, (d_2, t), r_2, q'') \in \delta$, then either $d_1 = d_2$, or $\phi(q, (c, t)) =$ true.

Note that the rules above restrict the transition relation δ and the pending function γ such that γ is unique. A command c is *enabled* in a state q for thread t if $\gamma(q, t) \in \{\perp, c\}$ (i.e., either no command is pending, or c itself is pending). A command c is *abort enabled* in a state q for thread t if c is enabled in q for thread t and there is no transition $(q, c, (d, t), r, q') \in \delta$ such that $d \in D$. Note that a transition $(q, c, (\text{abort}, t), 0, q') \in \delta$ from a state q can exist in two cases. First, if the command c is abort enabled for thread t in state q, which implies that the TM algorithm does not allow to continue the execution of command c for thread t in the state q. Second, if $\phi(q, (c, t)) =$ true, which implies that the TM algorithm can nondeterministically choose to abort the thread t in state q, if the command c is issued.

3.1 Contention managers

A *contention manager cm* on a set D of commands is a tuple $\langle P, p_{init}, \delta_{cm} \rangle$, where P is a set of states of the contention manager, $p_{init} \in P$ is the initial state of the contention manager, and $\delta_{cm} \subseteq P \times D \times P$ is the transition relation.

We now formalize a TM which uses a contention manager. Let a transactional memory M be represented by a TM algorithm $A = \langle Q, q_{init}, D, \phi, \gamma, \delta \rangle$. Let $cm = \langle P, p_{init}, \delta_{cm} \rangle$ be a contention manager. Then, M_{cm} is represented by a TM algorithm $A_{cm} = \langle Q_\times, (q_{init}, p_{init}), D, \phi_\times, \gamma_\times, \delta_\times \rangle$, where

- the set of states is $Q_\times = Q \times P$,
- the conflict function $\phi_\times$ is such that for all states $q_\times \in Q_\times$, for all commands $c \in C$, and for all threads $t \in T$, we have $\phi_\times(q_\times, (c, t)) = \phi(q, (c, t))$ where $q_\times = (q, p)$ for some state $p \in P$,
- the pending function $\gamma_\times$ is such that for all states $q_\times \in Q_\times$ and all threads $t \in T$, we have $\gamma_\times(q_\times, t) = \gamma(q, t)$ where $q_\times = (q, p)$ for some state $p \in P$,
- the transition relation $\delta_\times$ is such that for all states $q_\times, q'_\times \in Q_\times$, for all commands $c \in C$, for all statements $(d, t) \in \hat{S}_D$, and for all responses $r \in Resp$, we have $(q_\times, c, (d, t), r, q'_\times) \in \delta_\times$ if and only if

  (i)   there exists a transition $(q, c, (d, t), r, q') \in \delta$,
  (ii)   if $\phi(q, (c, t)) =$ true, then there exists a transition $(p, (d, t), p') \in \delta_{cm}$, and
  (iii) if there does not exist a transition $(p, (d, t), p') \in \delta_{cm}$, then $p = p'$, else $(p, (d, t), p') \in \delta_{cm}$,

  where $q, q' \in Q$ and $p, p' \in P$ such that $q_\times = (q, p)$ and $q'_\times = (q', p')$.

3.2 Languages of TM algorithms

A TM algorithm interacts with a scheduler. The scheduler chooses the next thread to be executed. A command of the chosen thread is given to the TM algorithm. The TM algorithm decides whether the command can be executed in a single or several atomic steps, or the command is in conflict. The TM algorithm makes a transition according to the transition relation, and gives back to the program a response. The response is $\perp$ if the TM algorithm needs additional steps to complete the command, 0 if the TM algorithm needs to abort the transaction of the scheduled thread, and 1 if the TM algorithm has completed the command. Given a scheduler and a TM algorithm, we get a set of runs. Projecting a run to the set of successful statements (that is, aborts, and statements that get response 1) gives a finite word. The language of a TM algorithm is the set of finite words that the TM algorithm can produce for any scheduler.

Formally, a *scheduler* σ on T is a function $\sigma : \mathbb{N} \to T$. A *run* $\rho = \langle q_0, c_0, (d_0, t_0), r_0 \rangle \ldots \langle q_n, c_n, (d_n, t_n), r_n \rangle$ of a TM algorithm A with scheduler σ is a finite sequence of tuples of states, commands, statements, and responses, where the following hold: (i) $q_0 = q_{init}$, and (ii) for all $j \geq 0$, there exists a transition $(q_j, c_j, (d_j, t_j), r_j, q_{j+1}) \in \delta$, and (iii) $t_j = \sigma(j)$. A statement $s_i = (d_i, t_i) \in \hat{S}$ is *successful* in the run $\rho = \langle q_0, c_0, s_0, r_0 \rangle \ldots \langle q_n, c_n, s_n, r_n \rangle$ if (i) $r_i \in \{0, 1\}$, or (ii) $r_k = 1$ with $i < k$ and for all j such that $i < j < k$, if $t_j = t_i$, then $r_j = \perp$. We define the *language* $L(A)$ of a TM algorithm A as the set of all finite words $w \in \hat{S}^*$ such that w is the sequence of all successful statements in a run of A with some scheduler. A TM algorithm

Algorithm 1 *getSequential*(*Status*, *c*, *d*, *t*, *r*)

> **if** $c = $ (read, v) **or** $c = $ (write, v) **then**
>> **if** $d = c$ **and** $r = 1$ **then**
>>> **if** $Status(u) = $ finished for all threads $u \neq t$ **then**
>>>> $Status(t) := $ started
>>>> **return** *Status*
>
> **if** $c = $ commit **then**
>> **if** $d = c$ **and** $r = 1$ **then**
>>> **if** $Status(u) = $ finished for all threads $u \neq t$ **then**
>>>> $Status(t) := $ finished
>>>> **return** *Status*
>
> **if** $d = $ abort **and** $r = 0$ **then**
>> **if** c is abort enabled in q for thread t **then**
>>> $Status(t) := $ finished
>>> **return** *Status*
>
> **return** $\perp$

A defines a TM M if every finite prefix w of every word in M is in $L(A)$, and every word w in $L(A)$ can be extended to an infinite word in M.

3.3 TM examples

We now describe different transactional memories as TM algorithms. To keep our first example simple, we describe a sequential TM.

3.3.1 The sequential TM

The sequential TM executes the transactions sequentially (as ideally suited for a uniprocessor). We do not use a contention manager for the sequential TM, and hence set the conflict function to be always false. We define the *sequential TM algorithm* A_{seq} as $\langle Q, q_{init}, D, \phi, \gamma, \delta_{seq} \rangle$. A state $q \in Q$ is defined as a function $Status : T \rightarrow \{$finished, started$\}$. The initial state is $q_{init} = Status_0$, such that for all threads $t \in T$, we have $Status_0(t) = $ finished. The set of extended commands is $D = C$. For all states q and all statements (c, t), the conflict function $\phi(q, (c, t)) = $ false. The transition relation δ_{seq} is obtained using the procedure *getSequential* shown in Algorithm 1. For all states $q \in Q$, all commands $c \in C$, all extended commands $d \in D \cup \{$abort$\}$, all threads $t \in T$, and all responses $r \in Resp$, we have

- if *getSequential*$(q, c, d, t, r) = \perp$, then there does not exist a state $q' \in Q$ such that $(q, c, (d, t), r, q') \in \delta_{seq}$, and
- if *getSequential*$(q, c, d, t, r) = q'$ for some state $q' \in Q$, then $(q, c, (d, t), r, q') \in \delta_{seq}$.

For all TM examples we present in this section, we use a similar notation.

Algorithm 2 *get2PL*($\langle rs, ws \rangle$, c, d, t, r)

> **if** c is not enabled in q for thread t **then return** $\perp$
> **if** $c = $ (read, v) **then**
>> **if** $d = c$ **and** $r = 1$ **and** $v \in ws(t) \cup rs(t)$ **then**
>>> **return** $\langle rs, ws \rangle$
>>
>> **if** $d = $ (rlock, v) **and** $r = \perp$ **then**
>>> **if** $v \notin ws(u)$ for all threads $u \neq t$ **then**
>>>> $rs(t) := rs(t) \cup \{v\}$
>>>> **return** $\langle rs, ws \rangle$
>
> **if** $c = $ (write, v) **then**
>> **if** $d = c$ **and** $r = 1$ **and** $v \in ws(t)$ **then**
>>> **return** $\langle rs, ws \rangle$
>>
>> **if** $d = $ (wlock, v) **and** $r = \perp$ **then**
>>> **if** $v \notin (ws(u) \cup rs(u))$ for all threads $u \neq t$ **then**
>>>> $ws(t) := ws(t) \cup \{v\}$
>>>> **return** $\langle rs, ws \rangle$
>
> **if** $c = $ commit **then**
>> **if** $d = c$ **and** $r = 1$ **then**
>>> $rs(t) := \emptyset$; $ws(t) := \emptyset$
>>> **return** $\langle rs, ws \rangle$
>
> **if** $d = $ abort **and** $r = 0$ **then**
>> **if** c is abort enabled in q for thread t **then**
>>> $rs(t) := \emptyset$; $ws(t) := \emptyset$
>>> **return** $\langle rs, ws \rangle$
>
> **return** $\perp$

3.3.2 The two-phase locking TM

Our second TM example 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 acquired locks during the commit. A shared lock is acquired for reading, and an exclusive lock is acquired for writing. We do not use a contention manager with two-phase locking, and hence define the conflict function to be always false.

We define the *2PL TM algorithm* A_{2PL} as $\langle Q, q_{init}, D, \phi, \gamma, \delta_{2PL} \rangle$. A state $q \in Q$ is represented as the pair $\langle rs, ws \rangle$, where $rs : T \rightarrow 2^V$ is the shared lock set, and $ws : T \rightarrow 2^V$ is the exclusive lock set. The initial state $q_{init} = \langle rs_0, ws_0 \rangle$, where for all threads $t \in T$, we have $rs_0(t) = ws_0(t) = \emptyset$. The set of extended commands is $D = C \cup (\{$rlock, wlock$\} \times V)$. For all states q and all statements (c, t), the conflict function $\phi(q, (c, t)) = $ false. The transition relation δ_{2PL} is obtained using the procedure *get2PL* shown in Algorithm 2.

3.3.3 The dynamic software transactional memory

Dynamic software transactional memory (DSTM) [19] is one of the most popular transactional memories. DSTM faces a conflict when a transaction wants to own a variable which is owned by another thread. DSTM is our first example that uses a contention manager. Thus, we identify pairs of states and statements that lead to a conflict, and set the conflict function as true at those places.

Algorithm 3 $getDSTM(\langle Status, rs, os\rangle, c, d, t, r)$

> **if** c is not enabled in q for thread t **then return** $\bot$
> **if** $Status(t) =$ aborted **and** $d \neq$ abort **then return** $\bot$
> **if** $c = ($read$, v)$ **then**
> > **if** $d = c$ **and** $r = 1$
> > > **if** $v \in os(t)$ **then**
> > > > **return** $\langle Status, rs, os\rangle$
> > >
> > > **if** $v \notin os(t)$ **and** $Status(t) =$ finished **then**
> > > > $rs(t) := rs(t) \cup \{v\}$
> > > > **return** $\langle Status, rs, os\rangle$
>
> **if** $c = ($write$, v)$ **then**
> > **if** $d = c$ **and** $v \in os(t)$ **and** $r = 1$ **then**
> > > **return** $\langle Status, rs, os\rangle$
> >
> > **if** $d = ($own$, v)$ **and** $r = \bot$ **then**
> > > $os(t) := os(t) \cup \{v\}$
> > > **for all** threads $u \neq t$ such that $v \in os(u)$ **do**
> > > > $Status(u) :=$ aborted $rs(u) := \emptyset$ $os(u) := \emptyset$
> > >
> > > **return** $\langle Status, rs, os\rangle$
>
> **if** $c =$ commit **then**
> > **if** $d =$ validate **and** $r = \bot$ **and** $Status(t) =$ finished **then**
> > > $Status(t) :=$ validated
> > > **for all** threads $u \neq t$ such that $rs(t) \cap os(u) \neq \emptyset$ **do**
> > > > $Status(u) :=$ aborted $rs(u) := \emptyset$ $os(u) := \emptyset$
> > >
> > > **return** $\langle Status, rs, os\rangle$
> >
> > **if** $d = c$ **and** $r = 1$ **and** $Status(t) =$ validated **then**
> > > $Status(t) :=$ finished $rs(t) := \emptyset$ $os(t) := \emptyset$
> > > **for all** threads $u \neq t$ such that $rs(u) \cap os(t) \neq \emptyset$ **do**
> > > > $Status(u) :=$ invalid
> > >
> > > **return** $\langle Status, rs, os\rangle$
>
> **if** $d =$ abort **and** $r = 0$ **then**
> > $Status(t) :=$ finished $rs(t) := \emptyset$ $os(t) := \emptyset$
> > **if** c is abort enabled in q for thread t **then**
> > > **return** $\langle Status, rs, os\rangle$
> >
> > **if** $\phi(q, (c, t)) =$ true **then**
> > > **return** $\langle Status, rs, os\rangle$
>
> **return** $\bot$

Algorithm 4 $getTL2(\langle Status, rs, ws, ls, ms\rangle, c, d, t, r)$

> **if** c is not enabled in q for thread t **then return** $\bot$
> **if** $c = ($read$, v)$ **then**
> > **if** $d = c$ **and** $v \in ws(t)$ **and** $r = 1$ **then**
> > > **return** $\langle Status, rs, ws, ls, ms\rangle$
> >
> > **if** $d = c$ **and** $v \notin ws(t) \cup ms(t)$ **and** $r = 1$ **then**
> > > $rs(t) := rs(t) \cup \{v\}$
> > > **return** $\langle Status, rs, ws, ls, ms\rangle$
>
> **if** $c = ($write$, v)$ **then**
> > **if** $d = c$ **and** $r = 1$ **then**
> > > $ws(t) := ws(t) \cup \{v\}$
> > > **return** $\langle Status, rs, ws, ls, ms\rangle$
>
> **if** $c =$ commit **then**
> > **if** $d = ($lock$, v)$ **and** $r = \bot$ **then**
> > > **if** $Status(t) =$ finished **and** $v \in ws(t)$ **then**
> > > > $ls(t) := ls(t) \cup \{v\}$
> > > > **for all** threads $u \neq t$ such that $v \in ls(u)$ **do**
> > > > > $Status(u) :=$ aborted
> > > >
> > > > **return** $\langle Status, rs, ws, ls, ms\rangle$
> >
> > **if** $d =$ validate **and** $r = \bot$ **and** $Status(t) =$ finished **then**
> > > **if** $rs(t) \cap ms(t) = \emptyset$ **and** $ws(t) = ls(t)$ **then**
> > > > $Status(t) :=$ validated
> > > > **for all** threads $u \neq t$ such that $rs(t) \cap os(u) \neq \emptyset$ **do**
> > > > > $Status(u) :=$ aborted $rs(u) := \emptyset$ $os(u) := \emptyset$
> > > >
> > > > **return** $\langle Status, rs, ws, ls, ms\rangle$
> >
> > **if** $d = c$ **and** $r = 1$ **and** $Status(t) =$ validated **then**
> > > **for all** threads $u \neq t$ such that $rs(t) \cup ws(t) \neq \emptyset$ **do**
> > > > $ms(u) := ms(u) \cup ws(t)$
> > >
> > > $rs(t) := \emptyset$ $ws(t) := \emptyset$ $ls(t) := \emptyset$ $ms(t) := \emptyset$
> > > $Status(t) :=$ finished
> > > **return** $\langle Status, rs, ws, ls, ms\rangle$
>
> **if** $d =$ abort **and** $r = 0$ **then**
> > $Status(t) :=$ finished
> > $rs(t) := \emptyset$ $ws(t) := \emptyset$ $ls(t) := \emptyset$ $ms(t) := \emptyset$
> > **if** c is abort enabled in q for thread t **then**
> > > **return** $\langle Status, rs, ws, ls, ms\rangle$
> >
> > **if** $\phi(q, (c, t)) =$ true **then**
> > > **return** $\langle Status, rs, ws, ls, ms\rangle$
>
> **return** $\bot$

We define the *DSTM algorithm* A_{dstm} as $\langle Q, q_{init}, D, \phi, \gamma, \delta_{dstm}\rangle$. A state $q \in Q$ is defined as a 3-tuple $\langle Status, rs, os\rangle$, where $Status : T \rightarrow$ {aborted, validated, invalid, finished} is the status function, $rs : T \rightarrow 2^V$ is the read set, and $os : T \rightarrow 2^V$ is the ownership set. The initial state $q_{init} = \langle Status_0, rs_0, os_0\rangle$, where for all threads $t \in T$, we have $Status_0(t) =$ finished and $rs_0(t) = os_0(t) = \emptyset$. The set of extended commands is $D = C \cup (\{$own$\} \times V) \cup \{$validate$\}$. The conflict function $\phi(q, (c, t)) =$ true if and only if (i) $c = ($write$, v)$ and for some thread $u \neq t$ we have $v \in os(u)$, or (ii) $c =$ commit and $Status(t) =$ finished and for some thread $u \neq t$ we have $rs(t) \cap os(u) \neq \emptyset$. The transition relation δ_{dstm} is obtained using the procedure *getDSTM* shown in Algorithm 3.

We define an *aggressive contention manager* as $aggr = \langle\{p_{init}\}, p_{init}, \delta\rangle$ such that for all threads $t \in T$ and for all extended commands $d \in D$ such that $d \neq$ abort, we have $(p_{init}, (d, t), p_{init}) \in \delta$. Intuitively, the aggressive contention manager does not allow a transaction to abort itself in case of conflict.

3.3.4 The TL2 TM

Transactional locking 2 (TL2) [7] is a TM that uses global version numbers to ensure correctness. Version numbers allow efficient read set validation in a distributed setting. We model version numbers using modified sets for each thread. When a transaction commits, it adds its write set to the modified set of every thread with an unfinished transaction.

We define the *TL2 TM algorithm* M_{TL2} as the tuple $\langle Q, q_{init}, D, \phi, \gamma, \delta_{TL2}\rangle$. A state $q \in Q$ is defined as a 5-tuple $\langle Status, rs, ws, ls, ms\rangle$, where $Status : T \rightarrow$ {validated, finished, aborted}, $rs : T \rightarrow 2^V$ is the read set, $ws : T \rightarrow 2^V$ is the write set, $ls : T \rightarrow 2^V$ is the lock set, and $ms : T \rightarrow 2^V$ is the modified set. The initial state is given by $q_{init} = \langle Status_0, rs_0, ws_0, ls_0, ms_0\rangle$, where for all threads t, we have $Status_0(t) =$ finished, and $rs_0(t) = ws_0(t) = ls_0(t) = ms_0(t) = \emptyset$. The set of extended commands is $D = C \cup (\{$lock$\} \times V) \cup \{$validate$\}$. The conflict function $\phi(q, (c, t)) =$ true if and only if $c =$ commit and for some

$v \in ws(t)$ we have $v \in ls(u)$ for some thread $u \neq t$. The transition relation δ_{TL2} is obtained using the procedure *getTL2* shown in Algorithm 4.

We define a *polite contention manager* for the TL2 TM algorithm as $pol = \langle \{p_{init}\}, p_{init}, \delta \rangle$, where $\delta = \{(p_{init}, (\text{abort}, t), p_{init}) \mid t \in T\}$. Intuitively, the polite contention manager always requires a transaction to abort in case of conflict.

Table 1 shows runs with different schedules for each TM algorithm described above.

4 Reduction theorem for safety

We wish to prove the safety of TMs for any number of threads and variables. Also, the safety of a TM should not depend on the choice of the contention manager, i.e., a TM should be safe with any contention manager. Modeling contention managers explicitly in our formalism is not a feasible option. Contention managers may blow up the state space as the decisions of a contention manager may depend intricately on past behavior. For example, a simple random backoff contention manager, which asks a conflicting thread to back off for an arbitrarily long period of time would require an unbounded number of states. Moreover, we shall show that some of the structural properties break when we model a TM algorithm in conjunction with a particular contention manager.

We observe that a TM algorithm, without a contention manager, nondeterministically chooses a transition at the point of conflict. On the other hand, when the TM algorithm is used with a contention manager, the transition should exist in the transition relation of the TM algorithm and that of the contention manager. In other words, a contention manager restricts the set of runs of a TM algorithm. Thus, given a TM algorithm A and a contention manager cm, we have $L(A_{cm}) \subseteq L(A)$.

Thus, it is sufficient to prove the safety of a TM without a contention manager, in order to show that the TM using any contention manager is safe. We shall present a reduc-

tion theorem for strict serializability and opacity. The theorem states that if a TM ensures $(2, 2)$ strict serializability (resp. $(2, 2)$ opacity), then the TM ensures strict serializability (resp. opacity). The reduction theorem relies on certain structural properties of TMs.

We now define four structural properties for TMs. These properties are satisfied by sequential TM, two-phase locking TM, DSTM, and TL2 TM. For every property, we also explain why the mentioned TMs satisfy that property. Note that the properties are sufficient (and not necessary) conditions for the reduction theorem to hold.

Let M be a transactional memory, and let w be a finite prefix of a word in M.

P1. *Transaction projection.* Aborting and unfinished transactions can influence other transactions only by forcing them to abort. Thus, removing all aborting transactions and some of the unfinished transactions do not change the response of the TM to the remaining statements. Formally, let X be the set of transactions in w. We define the *transaction projection* of w on $X' \subseteq X$ as the subsequence of w that contains every statement of all transactions in X'. The property P1 states that the transaction projection of w on X', where X' contains all committing transactions, no aborting transactions, and any subset of the unfinished transactions in w, is in M. For instance, a TM satisfies P1 if for every thread t: (i) whenever a statement of an aborting or unfinished transaction of thread t changes the state of another thread u, then u cannot commit, and (ii) upon an abort, the state of t is reset to the initial state of t. All TMs (without contention managers) we know of satisfy P1. But, a TM with a contention manager that prioritizes transactions according to the number of times it has aborted in the past, does not satisfy the structural property of transactional projection. This is because, an abort of a transaction of thread t may be the reason why the next transaction of thread t commits.

P2. *Thread symmetry.* For non-overlapping transactions, the TM is oblivious to the identity of the thread executing the transaction. The property P2 states that if (i) w have no

Table 1 Examples of runs and words in the language of different TM algorithms

TM	Scheduler output	The sequence $s_0 s_1 \ldots$ in the run of $L(A)$	The word for the run of $L(A)$
seq	11122 …	$(r, 1)_1, (w, 2)_1, c_1, (w, 1)_2, c_2 \ldots$	$(r, 1)_1, (w, 2)_1, c_1, (w, 1)_2, c_2 \ldots$
	112122…	$(r, 1)_1, (w, 2)_1, a_2, c_1, (w, 1)_2, c_2 \ldots$	$(r, 1)_1, (w, 2)_1, a_2, c_1, (w, 1)_2, c_2 \ldots$
2PL	111112…	$(rl, 1)_1, (r, 1)_1, (wl, 2)_1, (w, 2)_1, c_1, (wl, 2)_2 \ldots$	$(r, 1)_1, (w, 2)_1, c_1 \ldots$
	1211112…	$(rl, 1)_1, a_2, (r, 1)_1, (wl, 2)_1, (w, 1)_1, c_1, (wl, 2)_2 \ldots$	$a_2, (r, 1)_1, (w, 2)_1, c_1 \ldots$
dstm	12211112…	$(r, 1)_1, (o, 1)_2, (w, 1)_2, (o, 2)_1, (w, 2)_1, v_1, c_1, a_2 \ldots$	$(r, 1)_1, (w, 1)_2, (w, 2)_1, c_1, a_2 \ldots$
	12222111…	$(r, 1)_1, (o, 1)_2, (w, 1)_2, v_2, c_2, (o, 2)_1, (w, 2)_1, a_1 \ldots$	$(r, 1)_1, (w, 1)_2, c_2, (w, 2)_1, a_1 \ldots$
TL2	112112212…	$(r, 1)_1, (w, 2)_1, (w, 1)_2, (l, 2)_1, v_1, (l, 1)_2, v_2, c_1, c_2 \ldots$	$(r, 1)_1, (w, 2)_1, (w, 1)_2, c_1, c_2 \ldots$
	11212122…	$(r, 1)_1, (w, 2)_1, (w, 1)_2, (l, 2)_1, (l, 1)_2, a_1, v_2, c_2 \ldots$	$(r, 1)_1, (w, 2)_1, (w, 1)_2, a_1, c_2 \ldots$

r read, w write, c commit, a abort, rl rlock, wl wlock, l lock, o own, v validate, and k chklock. Command (c, t) is written as c_t

aborting transactions, and (ii) there exist two threads u and t such that for all committing transactions x of u and y of t in the word w, 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. For instance, a TM satisfies P2 if (i) the state of a thread is set to its initial state upon a commit, and (ii) the transition relation is identical for all threads. All TMs we know of satisfy P2.

P3. *Variable projection.* If a transaction can commit, then removing all statements that involve some particular variables does not cause the transaction to abort. We define the *variable projection* of w on $V' \subseteq V$ as the subsequence of w that contains all commit and abort statements, and all read and write statements to variables in V'. The property P3 states that if w has no aborting transactions, then for all $V' \subseteq V$, the variable projection of w on V' is in M. For instance, a TM satisfies P3 if reading or writing a variable does not remove a conflict on other variables. All TMs we know of satisfy P3 as they track every variable accessed by every thread independently.

P4. *Monotonicity.* If a word is allowed by the TM, then more sequential forms of the word are also allowed. Formally, let $F \subseteq S^*$ be the set of opaque (resp. strict serializable) words with exactly one unfinished transaction. We define a function $seq : F \rightarrow 2^F$ such that if $w_2 \in seq(w_1)$ and y is the unfinished transaction in w_1, then (i) $com(w_2)$ is sequential and strictly equivalent to $com(w_1')$, and (ii) all statements of y in w_1' occur in w_2 in some order such that order of all conflicts of global reads in y with other transactions in w_1' is preserved, where w_1' is obtained from w_1 by adding for every transaction x that commits before y in w, a write of an auxiliary variable v_{xy} to x, and a read of v_{xy} to y. (These variables are introduced to maintain the order of transactions.) The monotonicity property for opacity (resp. strict serializability) states that if $w = w' \cdot s$, where $w' \in F$, and s is not an abort, and s is a statement of the unfinished transaction in w', then for every word $w_2 \in seq(w')$, the word $w_2 \cdot s$ is a finite prefix of a word in M. For instance, a TM satisfies P4 if it is unfinished commutative and commit commutative. A TM is *unfinished commutative* if for all words $w_p, w_q, w_s \in S^*$, if the word $w_p \cdot w_q \cdot s \cdot w_s$ is a finite prefix of a word in M, where s is a global read and no statement in w_q conflicts with s, then $w_p \cdot s \cdot w_q \cdot w_s$ is a finite prefix of a word in M. A TM is *commit commutative* if for all words $w_p, w_q, w_s \in S^*$, if $w_p \cdot w_q \cdot s \cdot w_s$ is a finite prefix of a word in M, where s is a commit of some transaction x and no statement in w_q conflicts with s, then the word $w_p \cdot x \cdot w_q' \cdot w_s$ is a finite prefix of a word in M, where w_q' is the word obtained by removing transaction x from w_q. These commutativity rules allow to make an interleaved word sequential. The TMs, sequential, 2PL, DSTM, and TL2 are unfinished commutative and commit commutative, and thus satisfy monotonicity.

Now, we shall use these four structural properties to prove the reduction theorem. The idea of the proof is as follows. We assume that a TM ensures the correctness property for two threads and two variables, but does not ensure the correctness property for more threads or variables. We start with an incorrect word in the language of the TM. We consider the shortest incorrect prefix of this word. We remove all aborting and all pending transactions except one pending transaction using the transaction projection property. Using the monotonicity property, we sequentialize this prefix. Using the thread symmetry property, we rename the word to be with two threads. Using the variable projection property, we get an incorrect word with two threads and two variables, which is in the language of the TM. This leads to a contradiction.

Theorem 1 *If a TM M ensures* $(2, 2)$ *strict serializability (resp.* $(2, 2)$ *opacity) and satisfies the properties* P1, P2, P3, *and* P4 *for strict serializability (resp. opacity), then M ensures strict serializability (resp. opacity).*

Proof We prove the theorem for strict serializability. A similar proof holds for opacity. The proof is by contradiction. Let $w \in M$ be not strictly serializable. Let w_p be the longest finite 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 P1, 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. 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 for strict serializability, there exists a word $w_p'' \in seq(w_p')$ such that the word $w_3 = w_p'' \cdot s$ is a finite prefix of a word in M. In w_3 only one transaction, x, does not execute sequentially. We note that the last statement of x is a commit. This is because strict serializability concerns only committed transactions, and the word w_p'' is strictly serializable while w_3 is not. Using property P2, 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. As w_4 is not strictly serializable, we know (by the definition of conflict) that one of the following holds: (i) $s_1 = ((\text{read}, v_1), t)$ and $s_2 = ((\text{read}, v_2), t)$ 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) $s_1 = ((\text{read}, v_1), t)$ 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 on two threads and two variables, by property P3. Also, we note that w_5 is not strictly serializable.

As we know that all words $w \in M$ on two threads and two variables are strictly serializable, we get a contradiction. $\square$

5 TM specifications for safety

Using the reduction theorem mentioned above, our safety verification problem reduces to checking the safety property for two threads and two variables. We now describe TM specifications for strict serializability and opacity. Suitable TM specifications can also be defined for stronger notions of safety, such as the notions described by Scott [25], by modifying the semantics of conflict.

Our verification technique relies on the fact that the TM specifications for strict serializability and opacity for two threads and two variables can be defined using a finite number of states. This is not obvious, as threads may be delayed arbitrarily, transactions may contain arbitrarily many statements and may be aborted arbitrarily often. The classical approach to checking whether a word is strictly serializable is to construct a directed graph $G = (V, E)$, called the conflict graph [22], of the committing transactions in the word. The conflict graph captures the precedence of the committing transactions based on the conflicts. Given a word $w = s_0 \ldots s_n$, 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 commits or aborts 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 if and only if the word w is strictly serializable. We note that the size of this construction is unbounded. The following parametrized word illustrates the point: $w_m =((\text{read}, v_1), t_1), (((\text{write}, v_1), t_2), (\text{commit}, t_2))^m, (\text{commit}, t_1)$. The number of vertices in the conflict graph of w_m is $m + 1$. Thus, we cannot aim to create a finite-state TM specification for strict serializability using conflict graphs.

We look at the issues we face in creating TM specifications for strict serializability and opacity.

Analysis of strict serializability. We look at two words and reason whether they are strictly serializable.

– Consider the word $w = ((\text{write}, v_1), t_2), ((\text{read}, v_1), t_1), ((\text{read}, v_2), t_3), (\text{commit}, t_2), ((\text{write}, v_2), t_1), ((\text{read}, v_1), t_3), (\text{commit}, t_1), (\text{commit}, t_3)$. The word w is illustrated in Fig. 1a. The transaction x has to serialize before y due to a conflict on v_1 (as x reads v_1 before y commits and y writes to v_1). Similarly, the transaction z has to serialize before x due to a conflict on v_2. However, z has to serialize after y due to a conflict on v_1 (z reads v_1 after v_1 is written and committed by y). So, w is not strictly serializable. On the other hand, if one of the transactions had not committed, the word would have been strictly serializable.

Fig. 1 Examples for strict serializability. The words are fragmented into transactions of different threads. We use the notation: w for write, r for read, c for commit, and a for abort

– Consider the word $w = ((\text{write}, v_1), t_2), ((\text{read}, v_2), t_2), ((\text{read}, v_3), t_3), ((\text{read}, v_1), t_1), (\text{commit}, t_2), ((\text{write}, v_2), t_3), ((\text{write}, v_3), t_1), (\text{commit}, t_1), (\text{commit}, t_3)$. The word is illustrated in Fig. 1b. The transaction x has to serialize before y due to a conflict on v_1. Similarly, the transaction z has to serialize before x due to a conflict on v_3. Also, z writes to the variable v_2 which is read by transaction y before z commits. Thus, z has to serialize after y. This makes w not strictly serializable.

These examples show that strict serializability is a property concerned with committing transactions.

Analysis of opacity. Designing a TM specification for opacity requires even further care. This is because even aborting transactions should be prevented from reading inconsistent values. To demonstrate the intricacies involved, we again give two examples.

– Consider the word $w = ((\text{write}, v_1), t_2), ((\text{read}, v_1), t_1), ((\text{read}, v_2), t_3), (\text{commit}, t_2), ((\text{write}, v_2), t_1), ((\text{read}, v_1), t_3), (\text{commit}, t_1)$. The word is illustrated in Fig. 2a. Transaction x has to serialize before y due to a conflict on v_1. Also, z has to serialize after y due to a conflict on v_1, and before x due to a conflict on v_2. Note that although

Fig. 2 Examples for opacity. The words are fragmented into transactions of different threads

z does not commit, opacity requires that transaction x does not commit. So, w is not opaque.

- Consider the word $w = (($write$, v_1), t_2), (($read$, v_1), t_1), ($commit$, t_2), (($read$, v_2), t_3), ($abort$, t_3), (($write$, v_2), t_1), ($commit$, t_1)$. The word is illustrated in Fig. 2b. Transaction x has to serialize before y due to a conflict on v_1. Transaction z has to serialize after y as they do not overlap in w. Also, z has to serialize before x due to the conflict on v_2. This makes w not opaque. This shows how a read of an aborting transaction may disallow a commit of another transaction, for the sake of opacity.

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 to handle unbounded delay between transactions, as committing transactions store the required information in the 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.

We now present TM specifications for strict serializability and opacity, and manually prove their correctness. Later, we give deterministic TM specifications, and use an antichain-based tool to prove that the language of deterministic TM specifications for two threads and two variables is indeed equivalent to that of the nondeterministic counterparts.

5.1 Nondeterministic specifications

Nondeterminism allows a natural construction of the TM specifications, where a transaction nondeterministically guesses a serialization point during its lifetime. A branch of the nondeterministic TM specification corresponds to a specific serialization choice of the transactions, which makes the construction simple and intuitive, though redundant.

Nondeterministic TM specification for strict serializability. The TM specification for strict serializability is based on the observation that *every committing transaction serializes at some point during its execution*. The TM specification makes a nondeterministic guess of when a transaction serializes. Depending upon the guess, the TM specification checks upon the commit of a transaction, whether the commit can be executed, or the transaction needs to abort.

Formally, we define the *nondeterministic TM specification for strict serializability* Σ_{ss} for n threads and k variables as the tuple $\langle Q, q_{init}, \delta_{ss} \rangle$. A state $q \in Q$ is a 6-tuple $\langle Status, rs, ws, prs, pws, sp \rangle$, where $Status : T \rightarrow \{$started, invalid, serialized, finished$\}$ is the status, $rs : T \rightarrow 2^V$ is the read set, $ws : T \rightarrow 2^V$ is the write set, $prs : T \rightarrow 2^V$ is the prohibited read set, $pws : T \rightarrow 2^V$ is the prohibited write set, and $sp : T \rightarrow 2^T$ is the serialization predecessor set for the threads. If $v \in prs(t)$ (resp. $v \in pws(t)$), then the status of the thread t is set to invalid if t globally reads (resp. writes to) v. A thread u is in the weak predecessor set of thread t if the unfinished transaction of u is a weak predecessor of the unfinished transaction of t. The initial state q_{init} is $\langle Status_0, rs_0, ws_0, prs_0, pws_0, sp_0 \rangle$, where $Status_0(t) =$ finished for all threads $t \in T$, and $rs_0(t) = ws_0(t) = prs_0(t) = pws_0(t) = sp_0(t) = \emptyset$ for all threads $t \in T$. We express the transition function δ_{ss} using the procedure *nondetSpec* shown in Algorithm 5. For all states $q \in Q$ and all statements $s \in \hat{S}$, the following hold: (i) if *nondetSpec*$(q, s, \pi_{ss}) = \bot$, then there is no state $q' \in Q$ such that $(q, s, q') \in \delta_{ss}$, and (ii) if *nondetSpec*$(q, s, \pi_{ss}) = q'$ for some state $q' \in Q$, then $(q, s, q') \in \delta_{ss}$.

Given a state q and a thread $t \in T$, the procedure *ResetState*(q, t) makes the following updates: (i) sets $Status(t)$ to finished, (ii) sets $rs(t), ws(t), prs(t), pws(t)$, and $sp(t)$ to $\emptyset$, and (iii) for all threads $u \neq t$, removes t from $sp(u)$.

Nondeterministic TM specification for opacity. Apart from the requirements of the above mentioned TM specification for strict serializability, opacity requires that even global reads of aborting transactions observe consistent values.

The nondeterministic TM specification for opacity is based on the observation that *every committing and aborting transaction should serialize at some point during its execution*. As for Σ_{ss}, the TM specification Σ_{op} makes

Algorithm 5 $nondetSpec(\langle Status, rs, ws, prs, pws, sp \rangle, s, \pi)$

---

  if $s = ((\text{read}, v), t)$ **then**
    if $v \in ws(t)$ **then return** $\langle Status, rs, ws, prs, pws, sp \rangle$
    if $Status(t) = \text{finished}$ **then**
      $sp(t) := \{u \in T \mid Status(u) = \text{serialized}\}$
      $Status(t) := \text{started}$
    $rs(t) := rs(t) \cup \{v\}$
    if $\pi = \pi_{op}$ **then**
      if $v \in prs(t)$ **then return** $\perp$
      for all threads $u \neq t$ **do**
        if $Status(u) = \text{serialized}$ and $t \notin sp(u)$ **then**
          if $v \in ws(u)$ **then**
            $Status(u) := \text{invalid}$
          else
            $pws(u) := pws(u) \cup \{v\}$
    if $\pi = \pi_{ss}$ **then**
      if $Status(t) = \text{serialized}$ and $v \in prs(t)$ **then**
        $Status(t) := \text{invalid}$
  if $s = ((\text{write}, v), t)$ **then**
    if $Status(t) = \text{finished}$ **then**
      $sp(t) := \{u \in T \mid Status(u) = \text{serialized}\}$
      $Status(t) := \text{started}$
    else if $Status(t) = \text{serialized}$ and $v \in pws(t)$ **then**
      $Status(t) := \text{invalid}$
    $ws(t) := ws(t) \cup \{v\}$
  if $s = (\text{commit}, t)$ **then**
    if $Status(t) \in \{\text{started}, \text{invalid}\}$ **then return** $\perp$
    for all threads $u \neq t$ **do**
      if $u \in sp(t)$ **then**
        $prs(u) := prs(u) \cup ws(t)$
        $pws(u) := pws(u) \cup rs(t) \cup ws(t)$
        if $(ws(u) \cap (ws(t) \cup rs(t))) \neq \emptyset$ **then**
          $Status(u) := \text{invalid}$
      if $u \notin sp(t)$ **then**
        if $ws(t) \cap rs(u) \neq \emptyset$ **then**
          $Status(u) := \text{invalid}$
    $ResetState(q, t)$
  if $s = (\varepsilon, t)$ **then**
    if $Status(t) \neq \text{started}$ **then return** $\perp$
    $Status(t) := \text{serialized}$
    $sp(t) := \{u \in T \mid Status(u) = \text{serialized}\}$
    if $\pi = \pi_{op}$ **then**
      for all threads $u \neq t$ **do**
        if $Status(u) = \text{started}$ **then**
          if $rs(u) \cap ws(t) \neq \emptyset$ **then** $Status(t) := \text{invalid}$
          $pws(t) := pws(t) \cup rs(u)$
        if $Status(u) = \text{serialized}$ **then**
          if $ws(u) \cap rs(t) \neq \emptyset$ **then** $Status(u) := \text{invalid}$
          $pws(u) := pws(u) \cup rs(t)$
  if $s = (\text{abort}, t)$ **then** $ResetState(q, t)$
  return $\langle Status, rs, ws, prs, pws, sp \rangle$

---

a nondeterministic guess of when a transaction serializes. Upon every global read and every commit of a transaction, Σ_{op} checks whether the command can be executed or the transaction needs to be aborted. The *nondeterministic TM specification for opacity* Σ_{op} is given by the tuple $\langle Q, q_{init}, \delta_{op} \rangle$. The set Q of states and the initial state q_{init} are identical to that of Σ_{ss}. The only difference comes in the transition relation δ_{op}. As for strict serializability, we obtain δ_{op} using the procedure *nondetSpec* with property π_{op}, instead of π_{ss}.

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

Proof We prove the theorem for strict serializability here. The TM specification Σ_{ss} for strict serializability guarantees by construction, that a transaction x does not commit iff one of the conditions, C1–C4, holds (graphically shown in Fig. 3):

C1. there exists a transaction y such that x serializes before y and y writes to a variable v and commits, and then x 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 TM specification Σ_{ss} makes a *guess* of when every committing transaction serializes. Depending upon the guess, each committing transaction follows certain restrictions on the commands which can be executed. Consider a run w of Σ_{ss}. Let X be the set of finished transactions in w. Let w' be the sequential word such that w' is transaction equivalent to w and for all transactions $x, y \in X$, we have $x <_{w'} y$ if the ε command of x comes before that of transaction y in w (Note that every non-empty transaction has the ε command exactly once.) Then, $com(w')$ is strictly equivalent to $com(w)$, as for every transaction $x \in X$, the transaction x commits in w only if none of the conditions C1–C4 holds for x. Hence, every word in $L(\Sigma_{ss})$ is strictly serializable.

Conversely, let w be strictly serializable. As w is strictly serializable, there is a sequential word w_s such that $com(w_s)$ is strictly equivalent to $com(w)$. Let the committing transactions in the sequential word w_s be given by the sequence $x_1 x_2 \ldots$ of transactions. We claim that w is a run of the TM

Fig. 3 The commits inside ovals are disallowed by the TM specification for strict serializability. Each condition shows various cases. The arrows represent different possible positions for a command to occur in a given condition. We write w for write, r for read, and c for commit. We write the statement $((w, v), t_k)$ as $(w, v)_k$. Thread t_1 executes transaction x and thread t_2 executes transaction y

specification Σ_{ss} such that for all i and j such that $i < j$, the transaction x_i serializes before x_j in the run. This is because (i) the TM specification nondeterministically 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 $x_1 x_2 \ldots$ that satisfies any of the conditions C1–C4, and commits in w. Thus, the word w is in the language $L(\Sigma_{ss})$. □

5.2 Deterministic specifications

In nondeterministic TM specifications, we consider a particular order of serialization of transactions in a given branch. This allows us to argue individually for different serialization orders, which in turn, allows us to locally reason for every pair of transactions. On the other hand, in a deterministic TM specification, we have to consider all possible serialization orders at the same time, which complicates the specification.

We use two different predecessor notions for creating deterministic TM specifications. We define that a transaction x is a *weak predecessor* of transaction y in a word w if y must serialize after x for both x and y to be committing transactions. Note that the relation, weak predecessor, is not a transitive relation. But, when a transaction y commits, all weak predecessors of y become weak predecessors of the transactions of which y is a weak predecessor. We say that a transaction x is a *strong predecessor* of transaction y in a word w if y must serialize after x in w. Unlike weak predecessor, strong predecessor is a transitive relation.

We now present the formal definitions of the deterministic TM specifications for strict serializability and opacity.

Deterministic TM specification for strict serializability. The *deterministic TM specification for strict serializability* Σ_{ss}^d is given by the tuple $\langle Q, q_{init}, \delta_{ss}^d \rangle$. A state $q \in Q$ is a 7-tuple $\langle Status, rs, ws, prs, pws, wp, sp \rangle$, where $Status : T \rightarrow \{started, invalid, pending, finished\}$ is the status, $rs : T \rightarrow 2^V$ is the read set, $ws : T \rightarrow 2^V$ is the write set, $prs : T \rightarrow 2^V$ is the prohibited read set, $pws : T \rightarrow 2^V$ is the prohibited write set, $wp : T \rightarrow 2^T$ is the weak predecessor set, and $sp : T \rightarrow 2^T$ is the strong predecessor set for the threads. If $v \in prs(t)$ (resp. $v \in pws(t)$), then the status of the thread t is set to invalid if t globally reads (resp. writes to) v. A thread u is in the weak predecessor set of thread t if the unfinished transaction of u is a weak predecessor of the unfinished transaction of t. The initial state q_{init} is $\langle Status_0, rs_0, ws_0, prs_0, pws_0, wp_0, sp_0 \rangle$, where $Status_0(t) = finished$ for all threads $t \in T$, and $rs_0(t) = ws_0(t) = prs_0(t) = pws_0(t) = wp_0(t) = sp_0(t) = \emptyset$ for all threads $t \in T$. We express the transition function δ_{ss}^d using the procedure $detSpec$ shown in Algorithm 6. The notation of $detSpec$ is similar to that of the procedure $nondetSpec$. Given a state q and a

thread $t \in T$, the procedure $ResetState(q, t)$ makes the following updates: (i) sets $Status(t)$ to finished, (ii) sets $rs(t), ws(t), prs(t), pws(t),$ $wp(t),$ and $sp(t)$ to $\emptyset$, and (iii) for all threads $u \neq t$, removes t from $wp(u)$ and $sp(u)$.

Deterministic TM specification for opacity. The deterministic TM specification for opacity builds upon the deterministic TM specification for strict serializability. The difference comes in the strong predecessor set. We exploit the relation of strong predecessors in such a way that even aborting transactions see consistent values. For example, if a thread u is a strong predecessor of t, and t is a weak predecessor of u, then u cannot commit but t can. Many similar cases of conflict have to be carefully considered to capture the exact notion of opacity. The *deterministic TM specification for opacity* Σ_{op}^d is given by the tuple $\langle Q, q_{init}, \delta_{op}^d \rangle$. The set of states and the initial state are the same as those for Σ_{ss}^d. Also, the transition relation δ_{op}^d can be similarly obtained from Algorithm 6 using the property π_{op} instead of π_{ss}.

5.3 Equivalence checking of nondeterministic and deterministic TM specifications

We build nondeterministic and deterministic TM specifications for two threads and two variables. We observe that the nondeterministic TM specifications presented are too large to be automatically determinized. However, surprisingly enough, the deterministic TM specifications we present turn out to be much smaller in size. Using an antichain-based tool [28], we establish that for two threads and two variables, the language of our deterministic TM specification for strict serializability (resp. opacity) is equivalent to the language of the nondeterministic specification for strict serializability (resp. opacity).

For strict serializability, our deterministic TM specification Σ_{ss}^d has only 3,520 states, whereas the nondeterministic one Σ_{ss} has 12,345 states. Similarly, for opacity, Σ_{op}^d has 2,272 states, while the nondeterministic specification Σ_{op} consists of 9,202 states. The antichain-based tool can prove both equivalences within 5 s. This leads us to the following theorem.

Theorem 3 $L(\Sigma_{ss}) = L(\Sigma_{ss}^d)$ *and* $L(\Sigma_{op}) = L(\Sigma_{op}^d)$.

5.4 Safety verification results

The reduction theorem for safety states that if we prove that an TM ensures (2, 2) strict serializability (resp. (2, 2) opacity), then the TM ensures strict serializability (resp. opacity). This in turn implies that the TM using any contention manager ensures strict serializability (resp. opacity). We now check the safety (strict serializability or opacity) of different TMs by checking whether the language of the TM algorithm

Algorithm 6 $det\,Spec(\langle Status, rs, ws, prs, pws, wp, sp\rangle, s, \pi)$

$\quad$ **if** $s = ((\mathsf{read}, v), t)$ **then**
$\quad\quad$ **if** $v \in ws(t)$ **then return** $\langle Status, rs, ws, prs, pws, wp, sp\rangle$
$\quad\quad$ **if** $\pi = \pi_{op}$ **then**
$\quad\quad\quad$ $U := \{u \in T \mid v \in prs(u)$ or $v \in prs(u')$ s.t. $u \in sp(u')\}$
$\quad\quad\quad$ **if** $t \in U$ **then return** $\bot$
$\quad\quad$ **if** $Status(t) = \mathsf{finished}$ **then**
$\quad\quad\quad$ $U := \{u \in T \mid Status(u) = \mathsf{pending}\}$
$\quad\quad\quad$ $U' := \{u' \in T \mid \exists u \cdot u' \in sp(u)$ and $Status(u) = \mathsf{pending}\}$
$\quad\quad\quad$ $wp(t) := wp(t) \cup U$
$\quad\quad\quad$ $sp(t) := sp(t) \cup U \cup U'$
$\quad\quad\quad$ $Status(t) := \mathsf{started}$
$\quad\quad$ $rs(t) := rs(t) \cup \{v\}$
$\quad\quad$ **if** $v \in prs(t)$ **then** $Status(t) := \mathsf{invalid}$
$\quad\quad$ **for all** threads $u \in T$ **do**
$\quad\quad\quad$ **if** $v \in ws(u)$ **then** $wp(u) := wp(u) \cup \{t\}$
$\quad\quad\quad$ **if** $v \in prs(u)$ **then** $wp(t) := wp(t) \cup \{u\}$
$\quad\quad$ **if** $\pi = \pi_{ss}$ **then return** $\langle Status, rs, ws, prs, pws, wp, sp\rangle$
$\quad\quad$ **for all** threads $u \in T$ such that $u = t$ or $t \in sp(u)$ **do**
$\quad\quad\quad$ $sp(u) := sp(u) \cup U$
$\quad\quad$ **for all** threads $u \in T$ such that $u \in sp(t)$ **do**
$\quad\quad\quad$ $pws(u) := pws(u) \cup \{v\}$
$\quad\quad\quad$ **if** $v \in ws(u)$ **then**
$\quad\quad\quad\quad$ $Status(u) := \mathsf{invalid}$
$\quad$ **if** $s = ((\mathsf{write}, v), t)$ **then**
$\quad\quad$ **if** $Status(t) = \mathsf{finished}$ **then**
$\quad\quad\quad$ $U := \{u \in T \mid Status(u) = \mathsf{pending}\}$
$\quad\quad\quad$ $U' := \{u' \in T \mid \exists u \cdot u' \in sp(u)$ and $Status(u) = \mathsf{pending}\}$
$\quad\quad\quad$ $wp(t) := wp(t) \cup U$
$\quad\quad\quad$ $sp(t) := sp(t) \cup U \cup U'$
$\quad\quad\quad$ $Status(t) := \mathsf{started}$
$\quad\quad$ $ws(t) := ws(t) \cup \{v\}$
$\quad\quad$ **if** $v \in pws(t)$ **then** $Status(t) := \mathsf{invalid}$
$\quad\quad$ **for all** threads $u \neq t$ **do**
$\quad\quad\quad$ **if** $v \in rs(u)$ **then**
$\quad\quad\quad\quad$ $wp(t) := wp(t) \cup \{u\}$
$\quad\quad\quad\quad$ **if** $\pi = \pi_{op}$ and $t \in sp(u)$ **then** $Status(t) := \mathsf{invalid}$
$\quad\quad\quad$ **if** $v \in pws(u)$ **then** $wp(t) := wp(t) \cup \{u\}$
$\quad$ **if** $s = (\mathsf{commit}, t)$ **then**
$\quad\quad$ **if** $t \in wp(t)$ **then return** $\bot$
$\quad\quad$ **if** $Status(t) = \mathsf{invalid}$ **then return** $\bot$
$\quad\quad$ **if** $\pi = \pi_{op}$ **then**
$\quad\quad\quad$ $U := \{u \mid u \in wp(t)$ or $u \in sp(u')$ for some $u' \in wp(t)\}$
$\quad\quad\quad$ **if** $t \in U$ **then return** $\bot$
$\quad\quad$ **for all** threads $u \in T$ such that $u \in wp(t)$ **do**
$\quad\quad\quad$ **if** $ws(u) \cap ws(t) \neq \emptyset$ **then** $Status(u) := \mathsf{invalid}$
$\quad\quad\quad$ **else** $Status(u) := \mathsf{pending}$
$\quad\quad\quad$ $prs(u) := prs(u) \cup prs(t) \cup ws(t)$
$\quad\quad\quad$ $pws(u) := pws(u) \cup pws(t) \cup ws(t) \cup rs(t)$
$\quad\quad\quad$ **for all** threads $u' \in T$ such that $t \in wp(u')$ **do**
$\quad\quad\quad\quad$ $wp(u') := wp(u') \cup \{u\}$
$\quad\quad\quad$ **for all** threads $u' \in T$ such that $ws(u') \cap ws(t) \neq \emptyset$ **do**
$\quad\quad\quad\quad$ $wp(u') := wp(u') \cup \{u\}$
$\quad\quad$ **for all** threads $u \in T$ such that $u = t$ or $t \in sp(u)$ **do**
$\quad\quad\quad$ $sp(u) := sp(u) \cup U$
$\quad\quad$ $ResetState(q, t)$
$\quad$ **if** $s = (\mathsf{abort}, t)$ **then** $ResetState(q, t)$
$\quad$ **return** $\langle Status, rs, ws, prs, pws, wp, sp\rangle$

---

is included in the language of the deterministic TM specification for the safety property. Table 2 shows our results and leads to the following theorem.

Table 2 Time for checking language inclusion for TM algorithms on a dual core 2.8 GHz PC with 2 GB RAM

TM	Size	$L(A) \subseteq L(\Sigma_{ss})$	$L(A) \subseteq L(\Sigma_{op})$
seq	3	Y, 0.01s	Y, 0.01s
2PL	99	Y, 0.01s	Y, 0.01s
dstm	1,846	Y, 0.16s	Y, 0.13s
TL2	21,568	Y, 3.2s	Y, 2.4s
mod*TL2$_{pol}$*	17,520	N, w_1, 9s	N, w_1, 8s
	Counterexamples		
w_1	$(w, 2)_1, (w, 1)_2, (r, 2)_2, (r, 1)_1, c_2, c_1$		

In case the language inclusion holds, we write Y followed by the time required for finding it. Otherwise, we write N followed by the counterexample produced, followed by the time required to find the counterexample

Theorem 4 *The sequential TM, two-phase locking TM, DSTM, and TL2 ensure opacity.*

Our tool discovered a subtle point in TL2. In the description of the published TL2 algorithm, we found the order of two operations, validating the read set (rvalidate), and checking whether a variable in the read set is locked (chklock), ambiguous. We refined the TL2 algorithm shown in Algorithm 4 such that the extended command validate executes as two separate atomic operations, chklock and rvalidate, where chklock happens after rvalidate. We call this new TM algorithm as the modified TL2 TM algorithm. We use the polite contention manager with the modified TL2 TM algorithm. We found that the language of the TL2 TM algorithm with the polite contention manager is not included in the language of the TM specification for strict serializability. We obtain a counterexample. In the published TL2 algorithm, the authors maintain the version number and the lock bit of every variable in the same memory word. This ensures that the two operations chklock and rvalidate execute atomically, and thus they can be executed in any order. So, our experiments discover that the correctness of TL2 is based on the subtle fact that either the version number and the lock bit have to be accessed atomically, or rvalidate has to occur after chklock.

6 Model checking liveness

Unlike the safety properties, the liveness properties guaranteed by a TM may depend on the contention manager used with the TM. This is because the decision of a contention manager may require a thread to wait for an arbitrarily long period of time, or may require a thread to abort any conflicting transaction. Thus, we need to prove the liveness property of an TM using a specific contention manager.

We use the formalism of TM algorithms to verify liveness properties of TMs. We define a *loop l* in a TM algorithm A as a finite word $s_0 \ldots s_m$ such that there exists a run $\langle q_0, c_0, s_0, r_0 \rangle \ldots \langle q_m, c_m, s_m, r_m \rangle$ of A such that $q_0 = q_m$.

Note that we defined obstruction freedom using a Streett condition in Sect. 2 as $\bigwedge_{t \in T} (\Box \Diamond (\mathsf{abort}, t) \rightarrow \Box \Diamond ((\mathsf{commit}, t) \lor \bigvee_{c \in \hat{C}, u \in T \setminus \{t\}} (c, u)))$.

Note that every word w that is not obstruction free violates at least one of the conjuncts of the Streett condition stated above. Each conjunct (Streett pair) corresponds to one thread. A word w can violate the condition for thread t, only if w has from some point on only statements of t. Note that in this case w trivially satisfies the Streett pairs for other threads. This fact allows us to use a simple model checking procedure, even though obstruction freedom is formally a Streett condition.

In particular, a TM defined by a TM algorithm A ensures obstruction freedom iff there is no loop l in A such that all statements in l are from the same thread, and l contains no commit, and l contains an abort. Similarly, a TM ensures livelock freedom iff there is no loop l in A such l contains no commit, and every thread that has a statement in l, has an abort 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 words with two threads and one variable to generalize the result to all words. For this purpose, we describe two more structural properties of TMs. These properties are again satisfied by all TMs that we have discussed. Let $w = w_1 \cdot w_2$ be an infinite word such that w is in TM M, and no unfinished transaction in w_1 has a statement in w_2, and all statements in w_2 are from the same thread, and there is no commit command in w_2. For $i \in \{1, 2\}$, let V_i be the variables accessed in w_i.

P5. *Transaction projection.* A thread t running in isolation (no interleaved step from other threads) shall abort repeatedly only if it conflicts with some unfinished transaction. As the number of threads is finite, and a thread can have at most one unfinished transaction, there are infinitely many aborts of t due to a particular thread. The property P5 states that (i) the word $w_1' \cdot w_2$ is in M, where w_1' is obtained by taking the transaction projection of w_1 on non-aborting transactions, and (ii) if w_1 has no aborting transactions and w_2 reads or writes only one variable, then there exists a word $w' = w_1'' \cdot w_2$ in M, where w_1'' is obtained by projecting w_1 to transactions of some thread t that has statements in w_1. For instance, a TM satisfies P5 if the state of a thread is reset to the initial state upon an abort command, and every variable accessed by every thread is tracked independently.

P6. *Variable projection.* A thread t running in isolation shall abort repeatedly only if some commands corresponding to some variables are not allowed. As the number of variables is finite, there are infinitely many aborts of t due to a particular variable. The property P6 states that (i) there exists a word $w_1 \cdot w_2' \in M$ such that w_2' is the variable projection of w_2 on $\{v\}$ for some variable $v \in V_2$, and (ii) if w_1 has no aborting transactions, then the word $w' = w_1' \cdot w_2$ is in M, where w_1' is the variable projection of w_1 on V_2. For instance, a TM satisfies P6 if the TM tracks every variable accessed by every thread independently.

Theorem 5 *If a TM M ensures $(2, 1)$ obstruction freedom and satisfies the properties P5 and P6, then M ensures obstruction freedom.*

Proof Let M be a TM that ensures $(2, 1)$ obstruction freedom but not (n, k) obstruction freedom for some arbitrary n and k. Let $w \in M$ be a word such that w is not obstruction free. As w is not obstruction free, it can be written in the form $w_1 \cdot w_2$, such that (i) no unfinished transaction in w_1 has a statement in w_2, and (ii) all statements in w_2 are from the same thread, and (iii) there is no commit instruction in w_2. Let $w_3 = w_1 \cdot w_2'$ be a word such that w_2' is the projection of w_2 on one variable v. Using the variable projection property (P6 (ii)), we have $w_3 \in M$. We take a word $w_4 = w_1' \cdot w_2'$ such that w_1' has no aborting transactions and w_1' is on v. Using transaction projection (P5 (i)) and variable projection (P6 (i)), we get $w_4 \in M$. We now take a word $w_5 = w_1'' \cdot w_2'$ such that all commands in w_1'' are from one thread. From transaction projection (P5 (ii)), we get $w_5 \in M$. As w_5 is not obstruction free and w_5 is a word on two threads and one variable, we get a contradiction. $\square$

6.2 Liveness verification results

We built a verification tool to check obstruction freedom and livelock freedom properties of TM algorithms. To check obstruction freedom, our tool tries to find a loop l in the TM transition system such that all statements in l are from the same thread, and l has no commit, and l 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. Similarly, to check livelock freedom, our tool tries to find a loop l in the TM transition system such that there is no commit in l, and every thread that has a statement in l, has an abort in l.

In this way, our tool provides a platform for TM designers to check which liveness properties are ensured. If the liveness property fails, then the tool provides feedback in the form of a word that represents a counterexample. Our results are shown in Table 3 and lead to the following theorem.

Theorem 6 *DSTM with the aggressive contention manager ensures obstruction freedom but does not ensure livelock*

Table 3 Results of model checking liveness on a dual core 2.66 GHz desktop PC with 2 GB RAM

TM algorithm	Obstruction freedom	Livelock freedom
seq	N, w_1, $0.1s$	N, w_1, $0.1s$
2PL	N, w_1, $0.1s$	N, w_1, $0.1s$
$dstm_{aggr}$	Y, $2s$	N, w_2, $0.2s$
$TL2_{pol}$	N, w_1, $0.4s$	N, w_1, $0.4s$
	Counterexamples	
w_1	a_1	
w_2	a_1, $(r, 1)_1$, $(o, 1)_1$, a_2, $(o, 1)_2$	

The notation is similar to Table 2. The time denotes the time required to prove a liveness property or find a counterexample. The counterexamples obtained are of the form $a \cdot b^{\omega}$. We write the looping part b here

freedom. The sequential TM and two-phase locking TM do not ensure obstruction freedom. TL2 with the polite contention manager does not ensure obstruction freedom.

7 Related work

There has been recent independent work on the formal verification of TMs [5]. Cohen et al. model checked TMs applied to programs with a small number of threads and variables against the strong safety criteria of Scott [25]. They do not offer a reduction theorem and do not consider liveness properties. Cohen et al. later extended their safety verification technique [6] to programs with both transactional and non-transactional operations.

Our construction of the TM specifications is related to the work of Fle and Roucairol [8]. They investigated the set of concurrent traces that are generated by a finite set of iterating transactions. They proved that the language consisting of all traces that are conflict equivalent to a sequential trace is regular. Their results cannot be applied in the presence of aborting transactions, as they require the transitivity of conflicts, which does not hold when transactions may abort.

There has been much research on the formal verification of relaxed memory models and cache-coherence protocols for modern multi-processors, e.g., [4,10,16,23]. In this work, the semantics of a shared memory is generally given by a *memory consistency model*, which defines the possible outcomes of executing a concurrent program.

8 Conclusion

We presented a new technique for verifying TM safety and liveness properties. The cornerstones of our technique are finite-state representations for the languages of strictly serializable and opaque executions, a theorem that reduces the general verification problem to one for 2 threads and 2 variables, and a model-checking tool for TMs. Our method applies to all TM protocols that satisfy certain structural properties, and we successfully verified opacity for two-phase locking TM, DSTM, and TL2, and the obstruction freedom of DSTM.

To verify the correctness of a new TM using our methodology, one would proceed as follows. First, one needs to manually express the TM as a transition system, and manually check that the structural properties hold for the TM. Then, our tool automatically checks the desired safety or liveness property.

8.1 Limitations

Currently, our framework does not apply when transactions help each other. For instance, we cannot model Fraser's STM [9] where threads help each other in order to ensure livelock freedom. Also, our liveness properties capture deterministic notions. It will be interesting to account for probabilistic means to deal with contention, such as random exponential backoff. We assumed that the commands in the extended alphabet, like read, write, validate, and commit, execute atomically. So, TM algorithms have to guarantee this level of atomicity to ensure correctness using our methodology. We have extended our verification technique to hardware level atomicity [13]. Also, currently our framework does not support non-transactional code and nested transactions.

References

1. Alur, R., McMillan, K.L., Peled, D.: Model-checking of correctness conditions for concurrent objects. Inf. Comput. **160**, 167–188 (2000)
2. Anderson, J.H., Kim, Y., Herman, T.: Shared-memory mutual exclusion: major research trends since 1986. Distrib. Comput. **16**, 75–110 (2003)
3. Browne, M.C., Clarke, E.M., Grumberg, O.: Reasoning about networks with many identical finite state processes. Inf. Comput. **81**(11), 13–31 (1989)
4. Burckhardt, S., Alur, R., Martin, M.M.K.: CheckFence: checking consistency of concurrent data types on relaxed memory models. In: PLDI, pp. 12–21 (2007)
5. Cohen, A., O'Leary, J., Pnueli, A., Tuttle, M.R., Zuck, L.: Verifying correctness of transactional memories. In: FMCAD, pp. 37–44 (2007)
6. Cohen, A., Pnueli, A., Zuck, L.D.: Mechanical verification of transactional memories with non-transactional memory accesses. In: CAV, pp. 121–134. Springer (2008)
7. Dice, D., Shalev, O., Shavit, N.: Transactional locking II. In: DISC, pp. 194–208. Springer (2006)
8. Flé, M., Roucairol, G.: Maximal serializability of iterated transactions. Theor. Comput. Sci. **38**(11), 1–16 (1985)
9. Fraser, K., Harris, T.: Concurrent programming without locks. ACM Trans. Comput. Syst. (2007)

10. Gopalakrishnan, G., Yang, Y., Sivaraj, H.: QB or Not QB: an efficient execution verification tool for memory orderings. In: CAV, pp. 401–413. Springer (2004)
11. Guerraoui, R., Henzinger, T.A., Jobstmann, B., Singh, V.: Model checking transactional memories. In: PLDI, pp. 372–382 (2008)
12. Guerraoui, R., Henzinger, T.A., Singh, V.: Completeness and non-determinism in model checking transactional memories. In: CONCUR, pp. 21–35 (2008)
13. Guerraoui, R., Henzinger, T.A., Singh, V.: Software transactional memory on relaxed memory models. In: CAV, pp. 321–336 (2009)
14. Guerraoui, R., Herlihy, M., Pochon, B.: Polymorphic contention management. In: DISC, pp. 303–323 (2005)
15. Guerraoui, R., Kapalka, M.: On the correctness of transactional memory. In: PPoPP, pp. 175–184 (2008)
16. Henzinger, T.A., Qadeer, S., Rajamani, S.K.: Verifying sequential consistency on shared-memory multiprocessor systems. In CAV, pp. 301–315. Springer (1999)
17. Herlihy, M.: Wait-free synchronization. ACM Trans. Program. Lang. Syst. **13**(1), 124–149 (1991)
18. Herlihy, M., Luchangco, V., Moir, M.: Obstruction-free synchronization: double-ended queues as an example. In: ICDCS, pp. 522–529. IEEE Computer Society (2003)
19. Herlihy, M., Luchangco, V., Moir, M., Scherer, W.N.: Software transactional memory for dynamic-sized data structures. In: PODC, pp. 92–101 (2003)
20. Herlihy, M., Moss, J.E.B.: Transactional memory: architectural support for lock-free data structures. In: ISCA, pp. 289–300. ACM Press (1993)
21. Larus, J.R., Rajwar, R.: Transactional Memory. Synthesis Lectures on Computer Architecture. Morgan & Claypool (2007)
22. Papadimitriou, C.H.: The serializability of concurrent database updates. J. ACM **26**(4), 631–653 (1979)
23. Qadeer, S.: Verifying sequential consistency on shared-memory multiprocessors by model checking. IEEE Transactions on Parallel and Distributed Systems, 730–741 (2003)
24. Scherer, W.N., Scott, M.L.: Advanced contention management for dynamic software transactional memory. In: PODC, pp. 240–248 (2005)
25. Scott, M.L.: Sequential specification of transactional memory semantics. In: TRANSACT (2006)
26. Shavit, N., Touitou, D.: Software transactional memory. In: PODC, pp. 204–213 (1995)
27. Streett, R.S.: Propositional dynamic logic of looping and converse is elementarily decidable. Inf. Control **54**, 121–141 (1982)
28. De Wulf, M., Doyen, L., Henzinger, T.A., Raskin, J.-F.: Antichains: a new algorithm for checking universality of finite automata. In: CAV, pp. 17–30. Springer (2006)