Proving Opacity of a Pessimistic STM*

Simon Doherty[1], Brijesh Dongol[2], John Derrick[3], Gerhard Schellhorn[4], and Heike Wehrheim[5]

1 **Department of Computing, University of Sheffield, Sheffield, UK**
s.doherty@sheffield.ac.uk
2 **Department of Computer Science, Brunel University, London, UK**
Brijesh.Dongol@brunel.ac.uk
3 **Department of Computing, University of Sheffield, Sheffield, UK**
j.derrick@sheffield.ac.uk
4 **Universität Augsburg, Institut für Informatik, Augsburg, Germany**
schellhorn@informatik.uni-augsburg.de
5 **Universität Paderborn, Institut für Informatik, Paderborn, Germany**
wehrheim@upb.de

Abstract

Transactional Memory (TM) is a high-level programming abstraction for concurrency control that provides programmers with the illusion of atomically executing blocks of code, called transactions. TMs come in two categories, optimistic and pessimistic, where in the latter transactions never abort. While this simplifies the programming model, high-performing pessimistic TMs can be complex. In this paper, we present the first formal verification of a pessimistic software TM algorithm, namely, an algorithm proposed by Matveev and Shavit. The correctness criterion used is *opacity*, formalising the transactional atomicity guarantees. We prove that this pessimistic TM is a refinement of an intermediate opaque I/O-automaton, known as TMS2. To this end, we develop a *rely-guarantee* approach for reducing the complexity of the proof. Proofs are mechanised in the interactive prover Isabelle.

1998 ACM Subject Classification D.2.4 Software/Program Verification, F.1.2 Modes of Computation, F.3.1 Specifying and Verifying and Reasoning about Programs, H.2.4 Concurrency

Keywords and phrases Pessimistic STMs, Opacity, Verification, Isabelle, Simulation, TMS2

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2016.35

1 Introduction

Transactional memory (TM) is a mechanism that provides an illusion of atomicity in concurrent programs. It aims to reduce the burden on programmers of implementing complicated, error-prone synchronization mechanisms. TMs are analogous to database transactions in the sense that both perform a series of updates to data in an all-or-nothing manner — if a transaction succeeds, all its operations succeed, and otherwise, it aborts and all its operations fail. Since the first proposal of a software transactional memory (STM) [28], a number of STM algorithms have been developed [16], and many have made their way into mainstream programming, e.g., the ScalaSTM library, a new language feature in Clojure that uses an STM implementation internally for all data manipulation, the G++ 4.7 compiler (which supports STM features directly in the compiler) and others.

* Doherty and Dongol are supported by EPSRC Grants EP/M017044/1 and EP/N016661/1, respectively. Wehrheim is supported by DFG grant WE2290/8-2.

Intuitively, the purpose of an STM is that the transactions appear to be executed sequentially, i.e., as if their sections of code were protected by locks. However, unlike conventional locking mechanisms, STMs typically allow multiple transactions to be executed concurrently. The desired atomicity property for STMs is *opacity* [14, 3], which requires that all transactions (including aborting transactions) agree on a single sequential history of committed transactions. From a verification perspective opacity proofs represent a challenge beyond correctness conditions such as linearizability [10] due to interleaving at the level of operations as well as transactions.

There are two categories of STM designs: *optimistic* and *pessimistic*. Optimistic STMs assume that conflicts are rare, and when a conflict occurs some transaction is aborted. Transactional aborts cause work to be wasted, and interact badly with operations that are immediately visible outside of a transaction (e.g., consuming input from a stream, or printing to a console). Pessimistic STMs guarantee that no transaction ever needs to abort, thereby avoiding these difficulties. This can be easily achieved at the cost of sacrificing concurrency. For example, it is simple to implement a pessimistic STM that prohibits concurrency between read-only and writing transactions (e.g., by using a read/write lock). However, because conflicts between transactions are rare, overall performance can be improved by allowing read-only transactions to execute concurrently with writers. Supporting this concurrency can involve significant additional complexity, and this additional complexity can make the problem of verifying pessimistic algorithms significantly more difficult.

A number of approaches have so far studied verification of STMs, none of them, however, a pessimistic STM[1]. Here, we present a fully mechanised proof of correctness (i.e., opacity) of a pessimistic STM algorithm, namely that by Matveev and Shavit given in [24]. It poses a significant verification challenge due to the subtle nature of the synchronisation techniques it uses. Particularly difficult is showing that opacity holds when a writing transaction commits (see Listing 4), which may synchronise with another committing writer and all active readers.

Our proof of opacity proceeds via showing refinement (more precisely, a forward simulation) between the STM algorithm and a high-level opaque specification. This follows a general scheme for showing opacity proposed in [9], which used a specification called *TMS2*. Since the development of the TMS2 specification, there has been just one example of its use in a refinement-style verification of opacity [19], where the (simpler) NoRec STM is verified. Here, we present its first application to a pessimistic STM. To this end, both the STM implementation and the abstract specification are given as I/O-automata. This allows us to leverage existing theories within the interactive prover Isabelle [26] for our mechanised proof. The proof of refinement – as usual – requires a large number of invariants, both about the shared and local data of transactions. These invariants need to be shown to be preserved by all operations of all transactions. In order to decrease the complexity associated with such cross-preservation proofs (which are similar to interference-freedom proofs of [27]), we introduce a *rely principle* for transactions into invariance proofs (similar to rely-guarantee reasoning [17]). This provides a systematic way of stating assumptions on transactions as well as proving invariants. The work in this paper shows that this rely principle can make refinement-based proofs scale, even for complex STMs. All of our proofs have been carried out in Isabelle and can be found online [8].

Our presentation of the Matveev-Shavit algorithm is more precise than the original, and resolves certain ambiguities in the original description. In particular, a naive interpretation of the original description would result in an algorithm that was not opaque.

[1] A discussion of related work can be found in the conclusion.

Listing 1 Initialisation.

```
(globalVersion = 1) and (lock = free) and (∀ loc • (version (loc) = 0)) and
(∀ t • (txnVersion(t) = Idle) and (not writerWaiting(t)) and
      (t.wrSet = {}) and (not t.progressSeen))
```

Listing 2 Reader transaction's operations.

```
1:  procedure READ_BEGIN (t)
2:      txnVersion(t) ← Reading                ▷ Inform others that reader t has started
3:      t.temp ← globalVersion                 ▷ Read the global version
4:      txnVersion(t) ← t.temp                 ▷ Set t's txnVersion to stored global version
5:  procedure READ_READ (t, loc)
6:      READ_FROM_MEM (t, loc)                 ▷ Execute as procedure READ_FROM_MEM
7:  procedure READ_COMMIT (t)
8:      txnVersion(t) ← Idle                   ▷ Inform others that reader t has finished
9:  procedure READ_FROM_MEM (t, loc)
10:     if not t.progressSeen then             ▷ Check if committing writer's progress has been seen
11:         if version(loc) = txnVersion(t) then   ▷ Check if loc is potentially being written
12:             await txnVersion(t) ≠ globalVersion   ▷ Wait for committing writer to finish
13:             t.progressSeen ← true          ▷ Inform t's next READ_FROM_MEM that
                                                  the writer's commits have completed
14:     return mem(loc)                        ▷ Read value of loc from the memory
```

The structure of the paper is as follows. In Section 2, we introduce our running example and discuss the choices we made resolving the ambiguities in the original presentation of the algorithm. In Section 3, we introduce I/O automata as a model for opacity and the TMS2 specification. Section 4 develops our methodology based on refinement and rely-guarantee methods for proving opacity for pessimistic STMs. This is applied to the pessimistic STM of Matveev and Shavit [24] in Section 5. Finally, we conclude in Section 6.

2 A Pessimistic STM

In this section, we present the pessimistic STM by Matveev and Shavit [24] (which we will refer to as MSPessTM) where no transaction ever aborts. MSPessTM distinguishes between *read-only* (which perform no writes) and *write* transactions. A read-only transaction starts by calling `READ_BEGIN` , performs a number of `READ_READ` operations, then completes using the operation `READ_COMMIT` (see Listing 2). Similarly, a write transaction starts using operation `WRITE_BEGIN` , performs some number of reads and writes using `WRITE_READ` and `WRITE_WRITE` , respectively (Listing 3), then completes using `WRITE_COMMIT` (Listing 4).

Synchronisation is achieved using shared variables *globalVersion*, *txnVersion*(t) (t being a transaction), etc. as well as transaction-local variables $t.temp$, $t.progressSeen$ etc, which are initialised as in Listing 1. Some variables such as $t.temp$ are unrestricted initially, and hence, do not appear in Listing 1.

(1) MSPessTM uses a *deferred update* strategy: a write transaction t caches all its writes (pairs of locations *loc* and values v) in $t.wrSet$, which are committed to the shared memory when executing `WRITE_COMMIT` .

(2) Readers and writers are synchronised via the counter *globalVersion*. A committing writer will increment *globalVersion* prior to updating *mem* (with writes from its write set) and after these updates are completed. Thus, *globalVersion* is even iff there is a committing writer.

(3) After invoking `WRITE_BEGIN` , a writer transitions through three main phases: *waiting*, *active* and *committing*. There may be multiple waiting writers, but at most one active writer and at most one committing writer. Only the active writer may read or write, and only the committing writer may modify the shared memory. A waiting writer must not have progressed beyond line 18. A writer t becomes active when shared variable $writerWaiting(t)$ becomes *false* (either at line 19, or due to another writer executing line 39), and becomes committing by incrementing *globalVersion* (line 36).

(4) Synchronisation between writers is achieved as follows. Initially, there is neither an active nor a committing writer. Waiting writers compete for the shared *lock* (at line 18), and the winner becomes active. An active writer may enter the "critical section" for a committer by progressing beyond line 32 (which can only happen if there is no committer). The active writer actually becomes committing after executing line 36 (the first increment of *globalVersion*). At this point the writer is both active and committing, and only ceases to be an active writer after executing the code block in lines 37-41. Here, it either makes another writer active (line 39), or if no waiting writers are found, it simply releases *lock* (line 41). Matveev and Shavit refer to the mechanism at line 39 as "passing the baton" because *lock* is effectively transferred from the current active writer to some other waiting writer. Note that because lines 37-41 are executed after the first increment of *globalVersion*, there is no danger of there being more than one committing writer. A committing writer may need to synchronise with reads of another active writer; this is achieved using the mechanism described below.

(5) Synchronisation between readers and writers is the most complex mechanism of the algorithm. To understand this, we first note that from the perspective of a writer, there are two abstract versions of the memory: the *current memory* (which is the value of the shared *mem* variable) and the *new memory* (which is the *mem* updated with all writes in the write set). The synchronisation mechanisms ensure no transaction reads from both current memory and the new memory in an inconsistent manner. Note that a reader can read from the current and new memory without violating consistency if all of the locations read are unchanged (and MSPessTM allows this). Therefore, a writer distinguishes between *current* and *new readers*, which access the current and new memory, respectively.

A writer that has entered the critical section of `WRITE_COMMIT` (i.e., progressed beyond line 33) goes through four distinct phases: *blocking* new readers from accessing changed locations in the new memory (lines 34-35), waiting for *quiescence* from readers of the current memory (lines 42-43), *installing* the current memory (lines 44-45), and *signalling completion* (lines 46-47). Note that lines 36-41 deal with a writer committing then becoming inactive (but still committing) as described above.

A reader t must also wait if t detects that a new memory is being installed as the current memory (lines 10–12), which is true if the version number of the location *loc* that t wants to read is the same as t's transaction version. Such a reader must have read *globalVersion* after the first (but before the second) increment within `WRITE_COMMIT` . On the other hand, a writer waits for all readers that may be accessing the current memory during its quiescence phase. These are determined as non-idle transactions with a version number smaller than the writer's version number.

Listing 3 Writer transaction's begin, read and write operations.

```
15: procedure WRITE_BEGIN(t)
16:     writerWaiting(t) ← true                    ▷ Inform others that writer t has started
17:     while writerWaiting(t) do                  ▷ Check that t has become active
18:         atomic{if lock = free then lock ← taken else goto 17}   ▷ Try to acquire lock
19:         writerWaiting(t) ← false               ▷ t has acquired lock, so can become active
20:     t.temp ← globalVersion                     ▷ Read globalVersion
21:     txnVersion(t) ← t.temp                     ▷ Set txnVersion(t) to the globalVersion read

22: procedure WRITE_READ(t,loc)
23:     if loc ∈ dom(t.wrSet) then
24:         return t.wrSet(loc)                    ▷ If possible return value of loc from own write set
25:     else
26:         READ_FROM_MEM (t, loc)                 ▷ Otherwise return value of loc from the memory

27: procedure WRITE_WRITE(t,loc,v)
28:     t.wrSet ← t.wrSet ⊕ {loc ↦ v}              ▷ Store new value of loc in the write set
                                                   ▷ Notation f ⊕ {x ↦ a} denotes functional override
```

(6) A transaction t executing READ_FROM_MEM must wait for a committing writer at most once (line 12), i.e., after the current writer has committed, a reader will not need to wait for new active writers since any new active writer u is guaranteed to wait for the older reader t when u enters its quiescence phase. Hence, a variable $t.progressSeen$ is used to improve efficiency; once $t.progressSeen$ is set to *true*, future reads may safely read from memory without making further checks.

An intuitive English description of this algorithm is given in [24], but no more precise description is provided. In our work we have developed both a pseudocode description and a formal model of the algorithm. This has allowed us to resolve certain ambiguities in the original presentation. Our presentation is explicit about when the shared variables *globalVersion* and *txnVersion* need to be accessed. A direct implementation of the WRITE_COMMIT operation as presented in [24] would result in a procedure with several superfluous accesses to these variables, causing unnecessary and potentially inefficient memory activity. In the version of the algorithm that we verify, $txnVersion(t)$ is saved in the local variable $t.temp$ at line 30, and then this value is used throughout the operation.

Perhaps more importantly, Matveev and Shavit [24] do not detail how *globalVersion* should be copied into $txnVersion(t)$ at the beginning of a read-only transaction. A naive implementation that implemented this copy non-atomically (by first loading *globalVersion* into a local register and then writing the resulting value into $txnVersion(t)$) would not be opaque. To see this, consider the following execution: (1) some reader t_2 begins and reads *globalVersion* to a local register; (2) an already executing writer t_1 enters its commit operation and passes through the blocking phase (setting the version number of locations in its write set and incrementing *globalVersion*); and finally (3) t_1 checks for quiescence. Assuming that no other reader is currently active, t_1 sees quiescence *as it cannot detect that* t_2 *has already read globalVersion* and exits the loop in lines 42/43. If t_2 next executes the other half of the non-atomic statement, setting $txnVersion(t_2)$ to its old copy of *globalVersion*, we arrive at a situation where the reader t_2 can continue while the writer t_1 copies the values in $t_1.wrSet$ to the shared store. The fact that $txnVersion(t_2)$ is stale means that t_2 will be able to read from locations in $t_1.wrSet$ without becoming blocked at line 12 in READ_FROM_MEM , and may observe inconsistent values. We avoid this problem by using the special *Reading* value to

Listing 4 Writer transaction's commit operation.

```
29: procedure WRITE_COMMIT(t)
30:     t.temp ← txnVersion(t)        ▷ Load t's transaction version into a temporary variable
31:     if even(t.temp) then                                   ▷ Check for a committing writer
32:         await t.temp ≠ globalVersion                       ▷ Wait for committing writer to finish
33:         t.temp ← globalVersion                                    ▷ Re-read global version
34:     for all loc ∈ dom t.wrSet do                  ▷ Prepare to write each loc in t's write set
35:         version(loc) ← t.temp + 1          ▷ Inform other readers that loc is being updated
36:     globalVersion ← t.temp + 1 ▷ Update global version and become a committing writer
37:     if ∃ u • writerWaiting(u) then                          ▷ Check for a waiting writer
38:         choose t.txn ∈ {u | writerWaiting(u)}                ▷ Pick some waiting writer
39:         writerWaiting(t.txn) ← false              ▷ Make the selected waiting writer active
40:     else
41:         lock ← free                                  ▷ Free the lock if no waiting writers seen
42:     for all t.txn ∈ {u | t ≠ u and READING(t, u)} do
43:         await not READING(t, t.txn)) ▷ Wait for each potential reader of current memory
                                          to finish or signal that it will read the new memory
44:     for all (loc, v) ∈ t.wrSet do
45:         mem(loc) ← v ▷ Update memory with new value for each element in the write set
46:     globalVersion ← t.temp + 2                  ▷ Inform others that commits have finished
47:     txnVersion(t) ← Idle                          ▷ Inform others writer t has finished
48: function READING(t, u)
49:     return txnVersion(u) ≠ Idle and txnVersion(u) ≤ t.temp
```

indicate when a beginning transaction is copying *globalVersion* (see line 2 of READ_BEGIN). This technique is used explicitly in Matveev and Shavit's lock-eliding STM [1] to solve essentially the same problem.

3 Modelling STMs as Input/Output Automata

To show that the MSPessTM algorithm satisfies *opacity*, we prove that it is a *refinement* of an intermediate opaque I/O-automaton, known as TMS2. In this section we introduce I/O-automata (IOA) [22] and the TMS2 specification [9], then give examples of the (straightforward) IOA encoding of MSPessTM.

The correctness condition we need to prove about MSPessTM is opacity [14]. Overall opacity guarantees that committed transactions should appear as if they are executed atomically, at some unique point in time, and aborted transactions, as if they did not execute at all. Amongst other things, opacity also guarantees that all reads that a transaction performs are valid with respect to a single memory snapshot.[2] Opacity is formulated as a condition on histories, i.e. sequences of operations of transactions. In the following, we will use the term *trace* to stand for such sequences. When proving opacity of an STM, we thus need to show that all traces an STM allows are opaque.

We do not give a formal definition of opacity here because our proof does not make use of it directly. Instead, our proof strategy leverages two existing results from the literature:

[2] In addition, opacity provides meaning to aborted transactions, but because our case study MSPessTM is a pessimistic algorithm, we elide these details.

the definition of the TMS2 specification by Doherty *et al.* [9], and the mechanised proof that TMS2 is opaque by Luchangco *et al.* [23]. Using these results, trace refinement between MSPessTM and the TMS2 specification proves opacity of MSPessTM.

TMS2 is formalised using input/output automata [22], and hence, our formalisations also use IOA. Moreover, Müller [25] has mechanised the IOA theory (including its simulation rules) in Isabelle, which is now part of the standard Isabelle distribution [26]. As our objective is a mechanised proof using an interactive theorem prover, we thus chose to carry out our proofs within Isabelle. Overall, we obtain a fully mechanised verification of opacity for MSPessTM.

I/O automaton (IOA). An IOA is a labeled transition system P with a set of states Σ_P, a set of actions $acts(P)$ (partitioned into internal and external actions), a set of start states $start(P) \subseteq \Sigma_P$ and a transition relation $trans(P) \subseteq \Sigma_P \times acts(P) \times \Sigma_P$.

The TMS2 specification. The TMS2 specification is given in Figure 1. In IOAs, transitions are typically specified in an operational style: every IOA has a number of variables and transitions are formulated by giving a precondition and an effect of the transition stated in terms of these variables. For each transition, the first line in Figure 1 gives the action name. The transition is *enabled* if all its preconditions, given after the keyword Pre, hold in the current state. The state modifications (effect) of the transition are given as a number of assignments after the keyword Eff. In that, the index t refers to the transaction executing the operation.

The transitions of TMS2 are designed to capture the structural patterns common to most STM implementations defined in terms of read and write operations. The state of TMS2 therefore includes a $status_t$, which is 'notStarted' initially. The status enforces that each transaction must execute TMBegin, then some number of TMRead and TMWrite operations, and finally TMEnd.[3] The status is 'ready' in between reads and writes, and 'committed' after the end of the transaction (i.e., when it has committed). Since operations of different transactions may execute concurrently, the abstract specification splits executing an external operation into several steps, including an *invocation* and a *response*. For example, for TMRead, the external step $inv_t(\texttt{TMRead}(loc))$ represents the invocation when reading from location loc, and $resp_t(\texttt{TMRead}(v))$ represents a read returning with value v. In between, an STM implementing TMS2 must at some time determine the value it reads. In TMS2 this is represented by the internal step $\texttt{DoRead}_t(loc, n)$, which computes v by setting $status_t$ to $readResp(v)$. The internal actions of TMS2 (those prefixed by Do) correspond to the points at which operations "take effect".

Like opacity, TMS2 guarantees that transactions satisfy two critical requirements: (*R1*) all reads and writes of a transaction work with a *single consistent memory snapshot* that is the result of all previously committed transactions, and (*R2*) the *real-time order* of transactions is preserved.

To ensure (*R1*), the state of TMS2 includes $\langle mems(0), \ldots, mems(maxIdx)\rangle$, which is a sequence of all possible memory snapshots. Initially the sequence consists of one element, the initial memory $mems(0)$. Committing writer transactions append a new memory $newmem$ to this sequence (cf. $\texttt{DoCommitWriter}_t$), by applying the writes of the transaction to the last element $mems(maxIdx)$. To ensure that the writes of a transaction are not visible to other transactions before committing, TMS2 (like MSPessTM) uses a *deferred update* semantics:

[3] The full TMS2 specification [9] includes transitions for cancelling and aborting a transaction, which we do not present here, since we do not need them for our pessimistic algorithm.

$inv_t(\texttt{TMBegin})$
Pre: $status_t = \text{notStarted}$
Eff: $status_t := \text{beginPending}$
$beginIdx_t := maxIdx$

$resp_t(\texttt{TMBegin})$
Pre: $status_t = \text{beginPending}$
Eff: $status_t := \text{ready}$

$inv_t(\texttt{TMRead}(loc))$
Pre: $status_t = \text{ready}$
Eff: $status_t := \text{doRead}(loc)$

$resp_t(\texttt{TMRead}(v))$
Pre: $status_t = \text{readResp}(v)$
Eff: $status_t := \text{ready}$

$inv_t(\texttt{TMWrite}(loc, v))$
Pre: $status_t = \text{ready}$
Eff: $status_t := \text{doWrite}(loc, v)$

$resp_t(\texttt{TMWrite})$
Pre: $status_t = \text{writeResp}$
Eff: $status_t := \text{ready}$

$inv_t(\texttt{TMEnd})$
Pre: $status_t = \text{ready}$
Eff: $status_t := \text{doCommit}$

$resp_t(\texttt{TMEnd})$
Pre: $status_t = \text{commitResp}$
Eff: $status_t := \text{committed}$

$\texttt{DoCommitReadOnly}_t(n)$
Pre: $status_t = \text{doCommit}$
$\text{dom}(wrSet_t) = \varnothing$
$validIdx(t, n)$
Eff: $status_t := \text{commitResp}$

$\texttt{DoCommitWriter}_t$
Pre: $status_t = \text{doCommit}$
$rdSet_t \subseteq mems(maxIdx)$
Eff: $status_t := \text{commitResp}$
$mems := mems \oplus newmem$

$\texttt{DoRead}_t(loc, n)$
Pre: $status_t = \text{doRead}(loc)$
$loc \in \text{dom}(wrSet_t) \vee validIdx(t, n)$
Eff: **if** $loc \in \text{dom}(wrSet_t)$ **then**
$status_t := \text{readResp}(wrSet_t(loc))$
else $v := mems(n)(loc)$
$status_t := \text{readResp}(v)$
$rdSet_t := rdSet_t \oplus \{loc \to v\}$

$\texttt{DoWrite}_t(loc, v)$
Pre: $status_t = \text{doWrite}(loc, v)$
Eff: $status_t := \text{writeResp}$
$wrSet_t := wrSet_t \oplus \{loc \to v\}$

where
$$maxIdx \;\widehat{=}\; max(\text{dom}(mems))$$
$$newmem \;\widehat{=}\; \{maxIdx + 1 \mapsto (latestMem \oplus wrSet_t)\}$$
$$validIdx(t, n) \;\widehat{=}\; beginIdx_t \leq n \leq maxIdx \wedge rdSet_t \subseteq mems(n)$$

Figure 1 The transition relation of TMS2.

writes are stored locally in the transaction t's write set $wrSet_t$ and only published to the shared state when the transaction commits.

All reads in TMS2 must be consistent (i.e., occur from a single memory snapshot), therefore each transaction t keeps track of all its reads from memory in a read set $rdSet_t$. A read of location loc by transaction t checks that either loc was previously written by t itself (**then** branch of $\texttt{DoRead}_t(loc)$), or that all values read so far, including loc, are from the same memory snapshot n, where $beginIdx_t \leq n \leq maxIdx$ (predicate $validIdx(t, n)$ from the precondition, which must hold in the **else** branch). In the former case the value of loc from $wrSet_t$ is returned, and in the latter the value from $mems(n)$ is returned and the read set is updated. The read set of t is also validated when a transaction commits (cf. $\texttt{DoCommitReadOnly}_t$ and $\texttt{DoCommitWriter}_t$). Note that when committing, a read-only transaction may read from a memory snapshot older than $mems(maxIdx)$, but a writing transaction must ensure that all reads in its read set are from most recent memory (i.e., $mems(maxIdx)$), since its writes will update the memory sequence with a new snapshot.

To ensure (*R2*), if a transaction u commits before transaction t starts, then the memory that t reads from must include the writes of u. Thus, when starting a transaction (cf. $inv_t(\texttt{TMBegin})$), t saves the current last index of the memory sequence, $maxIdx$, into a local variable $beginIdx_t$. When t performs a read, the check $validIdx(t, n)$ ensures that that the snapshot $mems(n)$ used has $beginIdx_t \leq n$, which implies that the writes of u are included.

$inv_t(\texttt{TMWrite}(loc, v))$	$write\_write_t(28)$	$resp_t(\texttt{TMWrite})$
Pre: $status_t = Ready$	Pre: $status_t = pending(28, loc, v)$	Pre: $status_t = writeResp$
$t \in Writers$	Eff: $status_t := writeResp$	Eff: $status_t :=$ ready
Eff: $status_t :=$	$wrSet_t :=$	
$pending(28, loc, v)$	$wrSet_t \oplus \{loc \to v\}$	

Figure 2 Transitions of MSPessTM for operation `WRITE_WRITE`.

Encoding MSPessTM as an IOA. The state of the IOA representing MSPessTM contains local variables $status_t$, $wrSet_t$, $progressSeen_t$ and $temp_t$, and shared variable $writerWaiting_t$ and $txnVersion_t$ for each transaction t. Note that $status_t$ (with initial value *NotStarted*) is used to model control flow within each transaction, and hence, does not appear explicitly within the pseudocode in Listings 1-4. The state of the IOA also contains synchronisation variables *globalVersion* (which models the shared global version counter) and *lock* (which models the active writer lock). Finally, the IOA must make the shared memory explicit, thus the state includes two shared variables: *mem* (which maps locations to values) and *version* (which maps each location to a version number).

The IOA models execution by representing each atomic step of the MSPessTM algorithm (typically every line in the algorithm) as single IOA transition. As in TMS2, for each MSPessTM operation, the invocations and responses are external; all other lines of code map to internal actions.

Input arguments to an operation executed by transaction t are modelled as part of the $status_t$ variable. In particular, whenever t is executing an operation, the value of $status_t$ is of the form $pending(pc, \langle input\ values \rangle)$, where pc is the line number of the next step to be executed, and $\langle input\ values \rangle$ are the input arguments. Note that for MSPessTM, $\langle input\ values \rangle$ is *none* for the begin and end operations, a location *loc* for read operations, and a location *loc* and value v for write operations. As an example, Figure 2 shows the three transitions for the `WRITE_WRITE` operation from Listing 3: an invocation action $inv_t(\texttt{TMWrite}(loc, v))$, an internal action $write\_write_t(28)$ (corresponding to line 28 in the algorithm, hence the name), and a response action $resp_t(\texttt{TMWrite})$. The set *Writers* in the precondition of $inv_t(\texttt{TMWrite}(loc, v))$ is used to denote the set of writer transactions; we assume that this set is predetermined in some manner.

4 Verifying opacity as Input/Output automata refinement

We are now equipped with two IOA specifications, one for MSPessTM and one for TMS2. Of the latter we already know that its traces are opaque. Our next objective is to show that MSPessTM refines TMS2 from which opacity of MSPessTM follows. The standard way of verifying a refinement is to use a *forward simulation* between the implementation and the specification, as this allows one to verify the refinement in a stepwise manner. In this section we define *forward simulations*, and then develop a novel method for verifying some of the invariants that one needs as part of the proof of forward simulations. Details of how we apply this to the simulation proof between MSPessTM and TMS2 are given in Section 5.

4.1 Proving opacity via refinement.

To verify that pessimistic STM algorithms are opaque we verify that their IOA representations (in this case MSPessTM) are a *refinement* of TMS2. To define refinement formally we need some definitions. An *execution* of an IOA P is a sequence σ of alternating states and actions, beginning with a state in $start(P)$, such that for all states σ_i except the last,

$(\sigma_i, \sigma_{i+1}, \sigma_{i+2}) \in \mathit{trans}(P)$. A *reachable* state of P is a state appearing in an execution of P. An *invariant* of P is a predicate satisfied by all reachable states of P. A *trace* of P is any sequence of (external) actions obtained by restricting the actions of P to its external actions. The set of traces of P represents P's externally visible behaviour.

Refinement is a property between the visible behaviours of abstract an IOA A and a concrete implementation IOA C. In particular, we say C *refines* A iff every trace of C is also a trace of A. In our setting, each externally visible behaviour consists of a sequence of invoke and response events, including the input/output values of reads and writes.

We let $\mathit{external}(A)$ and $\mathit{internal}(A)$ denote the external and internal actions of IOA A, respectively. Writing $cs \xrightarrow{a}_C cs'$ for $(cs, a, cs') \in \mathit{trans}(C)$, we define:

▶ **Definition 1.** A *forward simulation* from a concrete IOA C to an abstract IOA A is a relation $R \subseteq \Sigma_C \times \Sigma_A$ such that each of the following holds.

Initialisation.

$$\forall\, cs \in \mathit{start}(C) \bullet \exists\, as \in \mathit{start}(A) \bullet R(cs, as)$$

External step correspondence.

$$\forall\, cs \in \mathit{reach}(C), as \in \mathit{reach}(A), a \in \mathit{external}(C), cs' \in \Sigma_C \bullet$$
$$R(cs, as) \wedge cs \xrightarrow{a}_C cs' \quad \Rightarrow \quad \exists\, as' \in \Sigma_A \bullet R(cs', as') \wedge as \xrightarrow{a}_A as'$$

Internal step correspondence.

$$\forall\, cs \in \mathit{reach}(C), as \in \mathit{reach}(A), a \in \mathit{internal}(C), cs' \in \Sigma_C \bullet$$
$$R(cs, as) \wedge cs \xrightarrow{a}_C cs' \Rightarrow$$
$$R(cs', as) \vee \exists\, as' \in \Sigma_A, a' \in \mathit{internal}(A) \bullet R(cs', as') \wedge as \xrightarrow{a'}_A as'$$

The conditions for forward simulation we use here are adapted from Lynch and Vaandrager [21]; our step correspondence conditions use a single abstract step instead of a full sequence as in [21], since this is simpler and sufficient for our proof.

We have proved in Isabelle that the existence of a forward simulation (in the sense given here) is sufficient to ensure trace inclusion (this follows fairly directly from a lemma in the I/O-automaton theory of [25]). Furthermore, a proof that all traces of TMS2 are opaque has been completed in the PVS interactive prover by Luchangco *et al.* [23]. Therefore, proving the existence of a forward simulation from the MSPessTM automaton to TMS2 is sufficent to prove opacity of MSPessTM.

4.2 Proving an Invariant with a Rely

The verification of an actual forward simulation for a specific STM algorithm turns out to depend critically on a complicated invariant. In order to manage this complexity, we have developed, in Isabelle, a scheme that allows us to decompose our invariant into simpler components, and prove that our invariant holds with the help of a *rely condition*. We now describe this scheme in the general case. In Section 5.2, we show how to apply this scheme to the MSPessTM algorithm.

To describe the scheme generically, fix an automaton P whose actions are indexed by transactions from a set T, as in *TMS*2 and *MSPessTM*. That is, we assume $\mathit{acts}(P) \subseteq \mathit{Act} \times T$, for some set Act of action names.

Further, assume we are given a *shared invariant*, $\mathit{sharedI} \subseteq \Sigma_P$, that describes an invariant of P's shared state, and *transaction invariants*, $\mathit{txnI}_t \subseteq \mathit{Act} \times \Sigma_P$, $t \in T$, that describe the relationship between each transaction's local state *upon enabledness of the action* $a \in \mathit{Act}$ and the automaton's shared state. The reason for incorporating actions in transaction invariants is that invariants for transactions typically consists of lots of cases, differentiating between

the different program locations of the transactions. Thus, a transaction invariant $txnI_t(a, s)$ can be read as "the property that holds when transaction t executes a in state s".

Our goal is to prove that the composition of the shared invariant and the transaction invariant is an invariant of P. Formally, we must prove that for all $s \in reach(P)$,

$$sharedI(s) \wedge \forall (a, t) \in acts(P) \bullet txnI_t(a, s) \tag{1}$$

Observe that to prove invariance of property (1), it is sufficient to prove the following four properties:

start. Invariant (1) is true initially, i.e., for all $s \in start(P)$, $sharedI(s)$ and $\forall (a, t) \in acts(P) \bullet txnI_t(a, s)$.

shared. The shared invariant is preserved. Formally, for all states s, s', actions a and transaction t, if $sharedI(s) \wedge txnI_t(a, s)$ and $s \xrightarrow{a,t} s'$ then $sharedI(s')$.

self. Each step of each transaction preserves its own invariant. Formally, for all states s, s', actions a, a' and transaction t, if $sharedI(s) \wedge txnI_t(a, s)$ and $s \xrightarrow{a,t} s'$ then $txnI_t(a', s')$.

cross. Each step of each transaction preserves the invariant of every other transaction. Formally, for all states s, s', actions a, a' and transactions t, u where $t \neq u$, if $sharedI(s) \wedge txnI_t(a, s) \wedge txnI_u(a', s)$ and $s \xrightarrow{a,t} s'$ then $txnI_u(a', s')$.

Unfortunately, the last proof obligation, **cross**, introduces substantial complexity in any verification based on invariants and simulation. To see this, observe that for each step of each transaction t, we must consider the effect of the step on *every* action of the transaction u. If we were to prove the noninterference property directly, we would need to discharge quadratically many proof obligations, one obligation for each pair of actions. We address this issue by introducing a *rely* condition, which describes the possible interference that a transaction may experience during its execution. This method reduces the number of proof obligations from quadratic to linear in the number of actions.

Roughly speaking, a rely condition is a relation over the states of an automaton that must preserve the invariant of each transaction, and that must abstract the transitions of each transaction. We say that a relation $rely_t \subseteq \Sigma_P \times \Sigma_P$, $t \in T$, is a *rely condition* of P, if the following conditions hold.

guar. Each transaction preserves the rely of every other transaction. Formally, for all states s, s', actions a and transactions t, u where $t \neq u$, if $sharedI(s) \wedge txnI_t(a, s)$ and $s \xrightarrow{a,t} s'$ then $rely_u(s, s')$.

rely. The rely must ensure each transaction's invariant. Formally, for all states s, s', actions a and transactions t, if $sharedI(s) \wedge txnI_t(a, s)$ and $rely_t(s, s')$ then $txnI_t(a, s')$.

It is straightforward to see that properties **guar** and **rely** together imply property **cross** above. Thus, we have the following theorem.

▶ **Theorem 2.** *If $sharedI$ and $txnI_t$ for each $t \in T$ satisfy properties **start**, **shared**, and **self**, and there is some $rely_t$ for each $t \in T$ satisfying properties **guar** and **rely**, then for all $s \in reach(P)$, we have $sharedI(s) \wedge \forall (a, t) \in acts(P) \bullet txnI_t(a, s)$.*

This theorem has been formalized and proved in our Isabelle development.

Note that unlike some other rely/guarantee schemes, our rely condition is not required to be reflexive or transitive. In some other schemes, the rely condition describes the interference from any number of environment steps. In our setting, the purpose of the rely condition is to ensure that every step of every other transaction preserves the relying transaction's invariant, so transitivity is unnecessary. As we shall see, for our proof of MSPessTM, the rely condition we use is not transitive.

Of course, standard rely/guarantee approaches also employ a guarantee condition. In a conventional setting, the guarantee condition of a component enables it to be composed with other components whose rely conditions are unknown when the first component is developed. So long as the guarantee of one component implies the rely of the other, the composition is sound. In our setting, no transaction is able to modify any state in the environment of the transactional memory system. Therefore, no transaction is capable of interfering with any other component, except the other transactions. Thus, no explicit guarantee is necessary. We require only that each step of each transaction preserves the rely of every other transaction.

5 Application to MSPessTM

In this section we apply our theory to the verification of the MSPessTM algorithm. As part of the proof, we introduce two auxiliary variables: $CWriter$ and $AWriter$ which keep track of the committing and active writers, respectively. If there is no committing writer, then $CWriter = \bot$, otherwise it has the transaction identifier of the committing transaction (similarly $AWriter$). Initially, we set $AWriter = CWriter = \bot$. $CWriter$ is updated to t when transaction t executes line 36, and to $\bot$ when t executes line 46. $AWriter$ is updated to t either when t acquires the lock at line 18, or when some other (active and committing) transaction sets $writerWaiting_t$ to false at line 39. $AWriter$ is set to $\bot$ when some committing transaction releases the lock at line 41.

5.1 The Simulation Relation

We first define a simulation relation R between the states of MSPessTM and TMS2. We use cs to denote a concrete state (i.e., the state of MSPessTM) and as to denote an abstract state (i.e., the state of TMS2). The value of variable v in cs is given by $cs.v$ (and similarly $as.v$). For reasons of space, it is not possible to describe the entire simulation relation, so we focus our attention on the most challenging and important aspect of our proof: showing that each read operation returns a legal value. It is through read operations that transactions actually observe the state of the memory. The full simulation relation may be viewed online [8].

First, it must be possible to identify particular indices of the memory sequence $mems$ (which is part of as) using the variables of cs. For our refinement proof, we must identify the last element in $mems$ (i.e. $maxIdx$ in Figure 1). Recall that in MSPessTM, each writer increments $globalVersion$ twice when it commits and that $globalVersion = 1$ initially. Thus, the total number of committed write transactions is $\lfloor cs.globalVersion/2 \rfloor$. Also, recall that in the initial state of TMS2 $mems$ has one element, and each committing writing transaction appends a new memory snapshot to $mems$. Our simulation captures this by requiring:

$$\lfloor cs.globalVersion/2 \rfloor = as.maxIdx\,. \tag{2}$$

We must ensure that some step of the MSPessTM read operation corresponds (c.f., Definition 1) to the $\texttt{DoRead}_t(n)$ step of TMS2, for some n. For any transaction t, this abstract read index n is determined by the value of $txnVersion_t$ after t has executed either line 4 or line 21. We let:

$$readIdx_t = \lfloor txnVersion_t/2 \rfloor\,. \tag{3}$$

The index $readIdx_t$ is defined throughout the interval between the response of the transaction t's begin operation, and the point during the commit operation when t sets $txnVersion_t$ to

Idle.[4] Our simulation relation specifies that the read set of each transaction is consistent with $as.mems(readIdx(cs, t))$ throughout the interval over which *readIdx* is defined. This allows us to prove that the precondition of $\texttt{DoRead}_t(n)$ is satisfied over this interval.

We must also show that each concrete read operation returns the correct value (i.e., is consistent with the value returned by the abstract read). That is, we need to show that when a transaction executes line 14 of `READ_FROM_MEM` reading from location *loc*, that the value returned is $as.mems(cs.readIdx_t)(loc)$. To achieve this, our simulation relation must relate the values of the concrete memory to values in the abstract memory sequence. There are three cases to consider. In the first, there is no committing writer and the concrete memory is equal to the latest abstract memory $as.mems(maxIdx)$. In the second, there is a committing writer, t, but the quiescence check has not been passed. Then the abstract $\texttt{DoCommitWriter}_t$ has already added $mem \oplus cs.wrSet_t$ to the end of the abstract memory sequence, so the current concrete memory is now $as.mems(maxIdx - 1)$. If the quiescence check has been passed, then current memory is no longer read by any transaction, so the simulation just needs to state the second property, which holds, even if some elements of the write set have been written to *mem* already. Formally we have:

$$cs.CWriter = \bot \wedge cs.mem = as.mems(maxIdx) \tag{4}$$

$$\begin{array}{l} cs.CWriter = t \wedge (\exists\, u \neq t \bullet \neg\ quiescent(u, cs)) \\ \wedge\ cs.mem = as.mems(maxIdx - 1) \wedge cs.mem \oplus cs.wrSet_t = as.mems(maxIdx) \end{array} \tag{5}$$

$$cs.CWriter = t \wedge (\forall\, u \neq t \bullet quiescent(u, x)) \wedge cs.mem \oplus cs.wrSet_t = as.mems(maxIdx) \tag{6}$$

where $quiescent(u, cs)$ holds, iff *globalVersion* is equal to the *effective transaction version* of transaction u. This version, denoted $effTxnVer(cs, u)$, is equal to $temp_u$, when $txnVersion_u = Reading$, equal to *globalVersion* when $txnVersion_u = Idle$ (an *Idle* transaction is quiescent), and equal to $txnVersion(u)$ otherwise. Note that *quiescent* is equal to the procedure `READING` returning false, except for *Idle* transactions, which need not be checked.

Using (4), (5) and (6) a transaction executing line 14 of `READ_FROM_MEM` (t, loc) returns the correct value in state *cs*, provided that we can guarantee the following two properties.

index. Either $cs.readIdx_t = as.maxIdx$ holds or both $cs.readIdx_t = as.maxIdx - 1$ and $cs.CWriter \neq \bot$ hold.

loc. if $cs.txnVersion_t = cs.globalVersion$ then $cs.CWriter = \bot$ or $loc \notin \mathrm{dom}(ws)$, where $ws = cs.wrSet_{cs.CWriter}$.

The simulation relation, together with **index** implies $cs.mem(loc) = as.mems(cs.readIdx_t)(loc)$ so long as *loc* is not in the write set of any committing transaction, which in turn follows from property **loc**.

Properties **index** and **loc** are proved using the following invariants and transaction invariants of MSPessTM.

inv1. In any state for which $txnVersion_t$ is defined and t is not the committing writer, $globalVersion - 2 \leq txnVersion_t \leq globalVersion$ and $txnVersion_t = globalVersion - 2 \Rightarrow CWriter \neq \bot$.

inv2. $CWriter = \bot$ iff *globalVersion* is odd.

txinv1. Whenever a transaction t is enabled to execute line 14 of the `READ_FROM_MEM` procedure, $txnVersion_t < globalVersion$ or $version(loc) \neq txnVersion_t$.

[4] Recall that $txnVersion_t$ is guaranteed to be in $\mathbb{N}$ throughout this interval.

$$cs'.CWriter = t \Leftrightarrow cs.CWriter = t \tag{7}$$

$$cs.CWriter = t \Rightarrow cs'.globalVersion = cs.globalVersion \wedge \tag{8}$$
$$cs'.mem = cs.mem \wedge cs'.version = cs.version \wedge$$
$$\forall\, u \neq t \bullet (quiescent(cs, u) \Rightarrow quiescent(cs', u))$$

$$cs.AWriter = t \Rightarrow cs'.AWriter = t \wedge cs'.lock = cs.lock \wedge cs'.version = cs.version \wedge \tag{9}$$
$$(cs.CWriter = \bot \Rightarrow cs'.CWriter = \bot)$$

$$cs'.CWriter = cs.CWriter \Rightarrow cs'.globalVersion = cs.globalVersion \tag{10}$$

$$cs'.CWriter \neq cs.CWriter \Rightarrow cs'.globalVersion = cs.globalVersion + 1 \wedge \tag{11}$$
$$(cs'.CWriter = \bot \Rightarrow effTxnVer(cs, t) = cs.globalVersion) \wedge$$
$$(cs'.CWriter \neq \bot \Rightarrow cs.CWriter = \bot)$$

$$cs'.txnVersion_t = cs.txnVersion_t \tag{12}$$

$$\forall\, l \bullet cs'.version(l) = cs.version(l) \vee cs'.globalVersion < cs'.version(l) \tag{13}$$

$$cs'.writerWaiting_t \neq cs.writerWaiting_t \Rightarrow cs.writerWaiting_t \wedge \tag{14}$$
$$\neg\ cs'.writerWaiting_t \wedge cs'.lock = taken \wedge cs.CWriter \neq \bot \wedge$$
$$cs'.AWriter = t \wedge even(cs.globalVersion)$$

Figure 3 Our rely condition is the conjunction of these assertions, along with assertions stating that the local variables of each transaction are not changed.

txinv2. Whenever a transaction t is enabled to execute any of the `WRITE_COMMIT` procedure after line 35 until line 46, we have for all $loc \in \mathrm{dom}(wrSet_t)$, $version(loc) = globalVersion$.

Property **index** follows from invariants **inv1** and **inv2**. Property **loc** follows from invariants **txinv1** and **txinv2**, and we use the generic approach described above to verify these in turn.

5.2 Verifying the invariants for MSPessTM

We now outline how we proved invariants **inv1** and **inv2**. The full invariant is too long to present in this report, but Isabelle source describing the invariant can be obtained from [8]. We focus our attention on the rely condition, and explain how to prove that our key invariants are preserved by this rely. Our rely condition is presented in Figure 3. Note that this rely $rely_t$ states the properties which the transaction t can assume to hold between current and next state while the other transactions $u \neq t$ execute.

We first consider invariant **txinv2**. The antecedent of this invariant is false until t completes the loop at lines 34-35, after which the consequent is true by the effect of that loop. At this point t is the active writer. Properties (9) and (8) of the rely condition describe which aspects of the shared state are stable when a writing transaction is either active or committing. Together, these properties ensure that while $t = AWriter$ or $t = CWriter$, $version$ does not change. Further, properties (9) and (7) ensure that the value of $AWriter$ and $CWriter$ are not changed by another transaction, implying stability of **txinv2**.

We turn now to invariant **inv1**. This invariant is established when the transaction t writes its $temp$ variable into $txnVersion_t$. Properties (12) and (10) of the rely condition ensure that the invariant is preserved over transitions where $cs'.CWriter = cs.CWriter$, because none of the relevant variables are changed. When $cs'.CWriter \neq cs.CWriter$, there are two possibilities, both of which are described by property (11) of the rely. If $cs'.CWriter \neq \bot$, then

$cs.CWriter \neq \perp$, so $cs.txnVersion_t \neq cs.globalVersion - 2$ by invariant **inv1**. The invariant follows easily. If $cs'.CWriter = \perp$, then $cs.txnVersion_t = cs.globalVersion$ by property (11) itself, and hence $cs'.txnVersion_t = cs'.globalVersion - 1$ holds, which preserves the invariant. (Note that property (11) is not transitive because it stipulates that *globalVersion* can only be incremented.)

For reasons of space, we have ignored the question of how we prove the **guar** property of Section 4.2 for our rely. We note only that the transition relation of MSPessTM ensures that when $cs'.CWriter \neq cs.CWriter$ and $cs.CWriter = \perp$, the transition is the step of the transaction $cs'.CWriter$ when it increments *globalVersion* the second time at line 46. MSPessTM has the invariant that at this point, the state is quiescent. The fact that during these steps, $cs.txnVersion_t = cs.globalVersion$ for all t follows from this quiescence.

This proof has been mechanized in Isabelle. This effort took around three weeks of full time work, including building the MSPessTM model and stating and proving the invariant and simulation relation. The proof uses Isabelle theories, including an Isabelle formalisation of the TMS2 automaton, that had already been developed by the authors as part of a larger transactional memory verification project.

6 Related work and conclusions

A number of approaches have so far studied verification of STMs, none of them – however – a pessimistic STM. The proposed techniques range from model checking approaches [12, 13] to interactive proofs [19]. A comprehensive survey of STM verification methods can be found in [18, 6]. Model checking (e.g., [4]) is generally not suitable for our aims of rigorously verifying algorithms against all possible executions. One promising approach is by Guerraoui *et al.* [12, 13], who present a method for model checking opacity using a reduction theorem that lifts opacity for two threads and two variables to opacity for an arbitrary number of threads and variables. However, their specifications do not consider the values that are read or written, and hence, the link to the definition of opacity in [15] is unclear. Moreover, as far as we are aware, the proof of their reduction theorem itself has not been mechanised.

Li *et al.* [20] have verified STM algorithms, however they show correctness against their own abstract specification. Lesani [18] developed a formal proof method for opacity by splitting opacity into a number of other conditions (*markability*). In spirit, this technique is similar to linearization proofs which rely on finding statements in the code which represent linearization points. Very recent work includes [2], which proved the $CaPR^+$ algorithm correct with respect to a notion called *conflict opacity*, which is a subset of opacity. Emmi *et al.* [11] describe a method for inferring invariants in order to prove strict serializability of TM algorithms. This simplifies a crucial task in mechanised proofs; similar techniques could be used for other correctness conditions, including opacity. The verification of TMs in the presence of non-transactional code is studied in [5].

In this paper, we presented a proof of opacity of the pessimistic STM of [24]. Our proof is based on refinement against the TMS2 specification, leveraging existing work that has mechanically verified TMS2 to be opaque [23]. This significantly improves on our previous work that inductively checks opacity [7]. Furthermore, we have developed and used a new generalised reasoning scheme for proving *transaction invariants* via *rely conditions*. The new proof scheme reduces the number of proof obligations from quadratic (with respect to the number of lines of code) to linear complexity.

References

1. Y. Afek, A. Matveev, and N. Shavit. Pessimistic software lock-elision. In M. K. Aguilera, editor, *DISC*, volume 7611 of *LNCS*, pages 297–311. Springer, 2012.
2. A. S. Anand, R. K. Shyamasundar, and S. Peri. Opacity proof for CaPR+ algorithm. In *ICDCN*, pages 16:1–16:4, New York, NY, USA, 2016. ACM. doi:10.1145/2833312.2833445.
3. H. Attiya, A. Gotsman, S. Hans, and N. Rinetzky. A programming language perspective on transactional memory consistency. In P. Fatourou and G. Taubenfeld, editors, *PODC'13*, pages 309–318. ACM, 2013.
4. A. Cohen, J. W. O'Leary, A. Pnueli, M. R. Tuttle, and L. D. Zuck. Verifying correctness of transactional memories. In *FMCAD*, pages 37–44, Washington, DC, USA, 2007. IEEE Computer Society.
5. A. Cohen, A. Pnueli, and L. D. Zuck. Mechanical verification of transactional memories with non-transactional memory accesses. In A. Gupta and S. Malik, editors, *CAV*, volume 5123 of *LNCS*, pages 121–134. Springer, 2008.
6. A. Cristal, B. K. Ozkan, E. Cohen, G. Kestor, I. Kuru, O. S. Unsal, S. Tasiran, S. O. Mutluergil, and T. Elmas. Verification tools for transactional programs. In *Transactional Memory*, volume 8913 of *LNCS*, pages 283–306. Springer, 2015.
7. J. Derrick, B. Dongol, G. Schellhorn, O. Travkin, and H. Wehrheim. Verifying opacity of a transactional mutex lock. In *FM*, volume 9109 of *LNCS*, pages 161–177. Springer, 2015.
8. S. Doherty, B. Dongol, J. Derrick, G. Schellhorn, and H. Wehrheim. Isabelle files for a verification of a pessimistic STM algorithm. http://www.informatik.uni-augsburg.de/swt/projects/MSPessTM.html, 2016.
9. S. Doherty, L. Groves, V. Luchangco, and M. Moir. Towards formally specifying and verifying transactional memory. *Formal Asp. Comput.*, 25(5):769–799, 2013.
10. B. Dongol and J. Derrick. Verifying linearisability: A comparative survey. *ACM Comput. Surv.*, 48(2):19, 2015.
11. M. Emmi, R. Majumdar, and R. Manevich. Parameterized verification of transactional memories. *SIGPLAN Not.*, 45(6):134–145, June 2010.
12. R. Guerraoui, T. A. Henzinger, and V. Singh. Completeness and nondeterminism in model checking transactional memories. In F. van Breugel and M. Chechik, editors, *CONCUR*, pages 21–35. Springer, 2008.
13. R. Guerraoui, T. A. Henzinger, and V. Singh. Model checking transactional memories. *DISC*, 22(3):129–145, 2010.
14. R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *PPOPP*, pages 175–184, 2008.
15. R. Guerraoui and M. Kapalka. *Principles of Transactional Memory*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2010.
16. T. Harris, J. R. Larus, and R. Rajwar. *Transactional Memory, 2nd edition*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2010.
17. C. B. Jones. Specification and design of (parallel) programs. In *IFIP Congress*, pages 321–332, 1983.
18. M. Lesani. *On the Correctness of Transactional Memory Algorithms*. PhD thesis, UCLA, 2014.
19. M. Lesani, V. Luchangco, and M. Moir. A framework for formally verifying software transactional memory algorithms. In M. Koutny and I. Ulidowski, editors, *CONCUR 2012*, pages 516–530, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.
20. Y. Li, Y. Zhang, Y.-Y. Chen, and M. Fu. Formal reasoning about lazy-STM programs. *Journal of Computer Science and Technology*, 25(4):841–852, 2010.

21 N. Lynch and F. Vaandrager. Forward and backward simulations. *Information and Computation*, 121(2):214–233, 1995.

22 N. A. Lynch and M. R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *PODC*, pages 137–151, 1987.

23 V. Luchangco M. Lesani and M. Moir. Putting opacity in its place. In *WTTM*, 2012.

24 A. Matveev and N. Shavit. Towards a Fully Pessimistic STM Model. In *TRANSACT*, 2012.

25 O. Müller. I/O Automata and beyond: Temporal logic and abstraction in Isabelle. In J. Grundy and M. Newey, editors, *TPHOLs*, pages 331–348. Springer, 1998.

26 T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

27 S. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Inf.*, 6:319–340, 1976.

28 N. Shavit and D. Touitou. Software transactional memory. *DISC*, 10(2):99–116, 1997.