DOI 10.1007/s00165-017-0433-3

Formal Aspects of Computing

Formal Aspects of Computing

Mechanized proofs of opacity: a comparison of two techniques

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

[1] Department of Computing, University of Sheffield, Sheffield, UK
[2] Department of Computer Science, Brunel University, London, UK
[3] Institut für Informatik, Universität Augsburg, 86135 Augsburg, Germany
[4] Institut für Informatik, Universität Paderborn, 33098 Paderborn, Germany

Abstract. Software transactional memory (STM) provides programmers with a high-level programming abstraction for synchronization of parallel processes, allowing blocks of codes that execute in an interleaved manner to be treated as atomic blocks. This atomicity property is captured by a correctness criterion called *opacity*, which relates the behaviour of an STM implementation to those of a sequential atomic specification. In this paper, we prove opacity of a recently proposed STM implementation: the Transactional Mutex Lock (TML) by Dalessandro et al. For this, we employ two different methods: the first method directly shows all histories of TML to be opaque (proof by induction), using a linearizability proof of TML as an assistance; the second method shows TML to be a refinement of an existing intermediate specification called TMS2 which is known to be opaque (proof by simulation). Both proofs are carried out within interactive provers, the first with KIV and the second with both Isabelle and KIV. This allows to compare not only the proof techniques in principle, but also their complexity in mechanization. It turns out that the second method, already leveraging an existing proof of opacity of TMS2, allows the proof to be decomposed into two independent proofs in the way that the linearizability proof does not.

Keywords: Software transactional memory; Opacity; Verification; Refinement; KIV; Isabelle.

1. Introduction

Software transactional memory (STM) is a mechanism that provides an *illusion of atomicity* in concurrent programs and thus aims to reduce the burden of implementing synchronization mechanisms on a programmer. The analogy of STM is with database transactions, which perform a series of accesses/updates to data (via read and write operations) atomically in an all-or-nothing manner. Similarly with an STM, if a transaction succeeds, all its operations succeed, and otherwise, all its operations fail. Intuitively, an STM should behave like a lock mechanism for critical sections: transactions appear to be executed sequentially, but—unlike conventional locking mechanisms—STMs should (and do) allow for concurrency between transactions. STM implementations are

Correspondence and offprint requests to: B. Dongol, Brunel University, UK. E-mail: Brijesh.Dongol@brunel.ac.uk

Published online: 21 August 2017

increasingly finding their way into standard programming languages, e.g., the new class `StampedLock` of the Java 8 release (which uses STM-like features), 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) etc.

STM algorithms can be categorized in terms of their update mechanism. The first variant is *eager*. These implementations try to maximize throughput by allowing the read and write operations of a transaction to take effect immediately (*direct* updates). Concurrent read and write operation may conflict with each other and therefore, the variables accessed need to be protected, e.g., by locks. Some of the eager implementations also allow the shared data to become temporarily inconsistent when a conflict occurs. To restore consistency, at least one of the involved transactions is aborted and all of its effects (updates) are undone using rollback mechanisms. The second variant uses *deferred updates*. These implementations avoid conflicting changes by deferring single updates to one big compound update at the end of transaction, i.e., the commit operation. Read and write operations are collected throughout a transaction's execution without modifying the shared memory. At the commit operation, the collected reads and writes are validated for conflicts with other transactions. If a conflict occurs, the transaction aborts and no roll back is necessary. If no conflict occurs, memory updates become globally visible during the commit. The STM algorithm Transactional Mutex Lock (TML) [DDS+10], which we study in this paper uses eager updates.

An STM implementation always needs a careful fine-tuning between the objective of allowing concurrency (thereby gaining high performance) and that of guaranteeing atomicity (which potentially limits performance). To achieve a large degree of concurrency, most STM implementations employ fine-grained operations, not employing global locks. Hence, subtle errors arising only in specific interleavings of transactions are likely to occur but difficult to detect via, e.g., testing. This makes formal verification an indispensable ingredient of STM design.

The meaning of "correctness" of an STM implementation is open to interpretation. A correctness criterion for STMs broadly has to state the following property: every concurrent execution of transactions should behave like a sequential one. Formally, such executions are described as *histories*, i.e., sequences of events of invocations and responses of operations. One of the earliest correctness notions is *strict serializability* [Pap79], which stems from the database community. While it captures most of the desirable properties of an STM, like real-time order preservation and serialization of transactions, strict serializability does not define the behaviour of aborting transactions. Therefore, several new correctness conditions emerged to fill the gap, e.g., *opacity*[1] [GK10], *TMS1* and *TMS2* [DGLM13], and *virtual world consistency* [IR12]. Opacity requires all transactions (including aborting transactions) to agree on a single history of committed transactions, whereas TMS1 and virtual world consistency are weaker in that they allow different aborting transactions to observe different committed histories.

Attiya et al. [AGHR13] have shown that opacity is equivalent to a condition known as *observational refinement*, which guarantees refinement of clients when using an opaque STM implementation in place of an atomic specification.[2] Furthermore, opacity is known to imply TMS1 [LLM12b], i.e., any STM opaque implementation also satisfies TMS1. Therefore, the notion of correctness that we use in this paper is opacity [GK10], following the nomenclature of Attiya et al. [AGHR13] (under the assumption that our given set of histories is prefix closed).

Having decided on a notion of correctness, the next question is about the verification technique to be used. Our interest is in showing opacity for every possible usage of the STM, i.e., any number of processes executing all possible transactions. Intricate interplays between transactions in STMs easily lead to errors in manual proofs. For instance, Lesani and Palsberg showed that the DSTM and McRT algorithms (which were supposedly opaque) are actually not opaque [LP13]. Therefore, we aim for mechanized formal verification.

In this paper, we present two formal, mechanized proofs of correctness of the Transactional Mutex Lock (TML). As correctness criterion we employ the recently given definition of opacity of [GK10]. As described above, it provides strong guarantees to programmers in the form of *observational refinement* allowing programmers to reason about programs using opaque STMs in terms of atomic transactions.

Our first proof technique is fully mechanized within the interactive prover KIV [EPS+14] and leverages existing proof techniques [DSW11] for linearizability [HW90]. More specifically, the approach consists of two steps: we (1) show that all histories of TML are linearizable to histories in which first of all just reads and writes to memory

[1] There are numerous variations in the definition of opacity. The original definition [GK08] is not *prefix closed*, i.e., a history can be opaque, but one of its prefixes may not. This potentially allows unwanted anomalies, in this case the "early-read" anomaly, where a transaction is able to read and return a value prior to it being written by a committing transaction. This issue has been addressed in an updated version of opacity [GK10] by additionally requiring the original definition of opacity to hold for all prefixes of a history.

[2] Equivalence of TMS1 and observational refinement has also been proved [AGHR14], but this requires additional assumptions on STM implementations.

occur *atomically*, and (2) establish an invariant about such histories stating that they all have "matching" histories in which whole transactions are executed atomically. These two steps are necessary for covering the two sorts of non-atomicity in STMs: STMs decompose (atomic) transactions into several operations (begin, read, write etc.), but also further decompose these operations into several steps (accessing and manipulating so-called meta-data) as to allow for a maximum of concurrency. The former decomposition is accounted for in step (2), the latter in step (1). A preliminary version of this method was outlined in [DDS+15]. The present paper gives a more complete description of the proof and describes our mechanization.

Our second proof technique is also a fully mechanized method, in this case the proof has been done independently with with the interactive prover Isabelle [NPW02] and with KIV. We focus on the Isabelle proof, with some comparison in Sect. 5. In particular, we show that the TML implementation satisfies *TMS2* [DGLM13]. TMS2 has in turn been shown to implement opacity [LLM12b] using the PVS interactive theorem prover. Following Doherty et al., each correctness condition is defined as an *Input/Output Automaton (IOA)* [LV95] that only generates histories accepted by the condition in question. This readily gives us a formal specification, which can be used as the abstract level in a proof of refinement. Furthermore, the IOA framework is part of the standard Isabelle distribution. Our proof therefore proceeds by encoding TMS2 and TML as IOA within Isabelle, then proving the existence of a forward simulation, which in turn has been shown to ensure trace refinement of IOA [LV95]. Our second proof technique is closely related to that of Lesani et al. [LLM12a], who verify the NOrec algorithm [DSS10] using the PVS proof assistant [ORS92]. Both techniques have evolved from the same set of earlier work on linearizability verification [DGLM04]. Although the algorithms verified are different, it is possible to draw some comparisons between the Isabelle and PVS encodings of IOA and simulation proofs (see Sect. 6 for details).

In the last section of the paper, we compare and evaluate the two proof methods. We argue that the second proof method is simpler: it leverages the fact that traces of TMS2 are opaque, and does not require that we explicitly maintain a "matching" history in which entire transactions are executed atomically. We also identify various differences between the formalizations employed in each proof, and discuss the trade-offs involved.

Related work A comprehensive survey of STM verification methods can be found in [Les14, COC+15]; here we give a short overview. Model checking (e.g., [COP+07]) is generally not suitable for our aims of rigorously verifying algorithms against all possible executions. One promising approach is by Guerraoui et al. [GHS08, GHS10], 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 processes and variables). However, their specifications do not consider the values that are read/written, and hence, the link to the definition of opacity in [GK10], which requires a *memory semantics* is unclear. Moreover, as far as we are aware, the proof of their reduction theorem itself has not been mechanized.

Li et al. [LZCF10] have verified STM algorithms, however they show correctness against their own abstract specification. Lesani [Les14] 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 [ASP16], which proved the $CaPR^+$ algorithm correct with respect to a notion called *conflict opacity*, which is a subset of opacity. $CaPR^+$ is an STM algorithm with a rollback feature for aborting transactions. Emmi et al. [EMM10] describe a method for infering invariants in order to prove strict serializability of TM algorithms. This simplifies a crucial task in mechanized proofs; similar techniques could be used for other correctness conditions, including opacity.

Overview The paper is structured as follows: Section 2 gives an introduction to STM, presents our TML case study and defines the correctness criterion of opacity. In Sect. 3, we describe our first proof approach including a description of its application to TML. The second proof approach, again including its application to TML, is given in Sect. 4. We compare and discuss both approaches in Sect. 5. Section 6 concludes and discusses related work.

2. Software transactional memory and opacity

Software transactional memory (STM) provides programmers with an easy-to-use synchronisation mechanism for concurrent access to shared data. The basic mechanism is a programming construct that allows one to specify blocks of code as *transactions*, with properties of database transactions (e.g., atomicity, consistency and isolation) [HLR10]. All statements inside a transaction execute *as though they were atomic*. However—like database transactions—software transactions need not successfully terminate, i.e., might abort.

```
Init: glb = 0

TMBegin:                              TMEnd:
// B1 is LP if even glb               // E1 is LP if even loc
B1 do loc := glb                      E1 if (loc & 1)
B2 while (loc & 1)                    E2   glb := loc + 1;  // LP
B3 return ok;                         E3 return commit;

TMRead(addr):                         TMWrite(addr,val):
R1 val := *addr;                      W1 if (loc & 0)
R2 if (glb = loc) // LP               // W2 is LP when glb ≠ loc
R3   return val;                      W2   if (!cas(&glb, loc, loc+1))
R4 else return abort;                 W3      return abort;
                                      W4   else loc++;
                                      W5 *addr := val; // LP
                                      W6 return ok;
```

Fig. 1. The Transactional Mutex Lock (TML)

To support the concept of software transactions, STMs usually provide a number of operations to programmers: operations to start (`TMBegin`) or to end a transaction (`TMEnd`), and operations to read or write shared data (`TMRead`, `TMWrite`).[3] These operations can be called (invoked) from within a program (possibly with some arguments, e.g., the variable to be read) and then will return with a response. Except for operations that start transactions, all other operations might potentially respond with `abort`, thereby aborting the whole transaction. STMs expect the programmer to always start with `TMBegin`, then a number of reads and writes can follow, and eventually the transaction is ended by calling `TMEnd` unless one of the other operations has already aborted.

We present the TML algorithm in Sect. 2.1, formalise the concept of histories in Sect. 2.2, and give the informal and formal definitions of opacity in Sect. 2.3.

2.1. Example: Transactional Mutex Lock

In this paper, we will study a particular implementation of STMs, namely the Transactional Mutex Lock (TML) of Dalessandro et al. [DDS+10], which is given in Fig. 1. It provides exactly the four types of operations, but operation `TMEnd` in this algorithm will never respond with abort. Line numbers are denoted `B1` etc, and the references in the comments to LP are explained later.

TML adopts a very strict policy for synchronisation among transactions: as soon as one transaction has successfully written to a variable, all other transactions running concurrently will be aborted when they invoke another read or write operation. To this end, TML uses a global counter `glb` (initially 0) and local variable `loc`, which is used to store a copy of `glb`. Variable `glb` records whether there is a *live writing transaction*, i.e., a transaction which has been started, has not yet ended nor aborted, and has executed (or is executing) a write operation. More precisely, `glb` is odd if there is a live writing transaction, and even otherwise. Initially, we have no live writing transaction and thus `glb` is 0 (and hence even).

Operation `TMBegin` copies the value of `glb` into its local variable `loc` and checks whether `glb` is even. If so, the transaction is started, and otherwise, the process attempts to start again by rereading `glb`. A `TMRead` operation succeeds as long as `glb` equals `loc` (meaning no writes have occurred since the transaction began), otherwise it aborts the current transaction. The first execution of `TMWrite` attempts to increment `glb` using a `cas` (compare-and-swap), which atomically compares the first and second parameters, and sets the first parameter to the third if the comparison succeeds. If the `cas` attempt fails, a write by another transaction must have occured, and hence, the current transaction aborts. Otherwise `loc` is incremented (making its value odd) and the write is performed. Note that because `loc` becomes odd after the first successful write, all successive writes that are part of the same transaction will perform the write directly after testing `loc` at line $W1$. Further note that if the `cas` succeeds, `glb` becomes odd, which prevents other transactions from starting, and causes all concurrent live transactions still wanting to read or write to abort. Thus a writing transaction that successfully updates `glb` effectively locks shared memory. Operation `TMEnd` checks to see if a write has occurred by testing whether `loc` is odd. If the test succeeds, `glb` is set to `loc+1`. At line `E2`, `loc` is guaranteed to be equal to `glb`, and therefore this update has the effect of incrementing `glb` to an even value, allowing other transactions to begin.

[3] In general, arbitrary operations can be used here; for simplicity we use reads and writes to variables.

Table 1. Events appearing in the histories of TML

invocations	possible matching responses
$inv_p(\texttt{TMBegin})$	$res_p(\texttt{TMBegin(ok)})$
$inv_p(\texttt{TMEnd})$	$res_p(\texttt{TMEnd(commit)})$, $res_p(\texttt{TMEnd(abort)})$
$inv_p(\texttt{TMRead}(x))$	$res_p(\texttt{TMRead}(v))$, $res_p(\texttt{TMRead(abort)})$
$inv_p(\texttt{TMWrite}(x, v))$	$res_p(\texttt{TMWrite(ok)})$, $res_p(\texttt{TMWrite(abort)})$

The key question we want to answer in this paper is: "Does TML correctly implement an STM?". That is, does TML guarantee that transactions look as though they were executed atomically, even when a large number of transactions are running concurrently. Concurrently here means that the individual lines in the operations (i.e., B1, B2, etc) can be interleaved by different calling processes. We start by first fixing the meaning of "correctness" for an STM implementation as *opacity* [GK08], and this is defined in terms of *histories*. We thus begin by introducing some notation and definitions concerning histories in the next subsection, before defining opacity in Sect. 2.3.

2.2. Histories

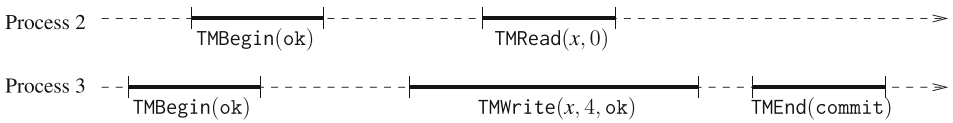
As standard in the literature, opacity is defined on the *histories* of an implementation. Histories are sequences of *events* that record all interactions between the implementation and its clients. Histories form an abstraction of the actual interleaving of individual lines of code, and thus an event is either an invocation (*inv*) or a response (*res*). For the TML implementation, possible invocation and matching response events are given in Table 1. In the table, p is a process identifier from a set of processes P (and is given as a subscript to an invocation or response), x is an address of a variable and v a value.

To define the semantics of a (sequential) history we model shared memory by a set L of addresses (or locations) mapped to values denoted by a set V. We will use the terms addresses and locations interchangeably. Hence the type $Mem \mathrel{\widehat{=}} L \to V$ describes the possible states of the shared memory. We assume that initially all addresses hold the value $O \in V$.

Example 2.1 In each of our examples we assume $V = \mathbb{Z}$ and that $O = 0$. The following history h_1 is a possible execution of the TML. It accesses the address x by two processes 2 and 3 running concurrently.

$$h_1 \mathrel{\widehat{=}} \langle inv_3(\texttt{TMBegin}), inv_2(\texttt{TMBegin}), res_3(\texttt{TMBegin(ok)}), res_2(\texttt{TMBegin(ok)}), inv_3(\texttt{TMWrite}(x, 4)), inv_2(\texttt{TMRead}(x)), res_2(\texttt{TMRead}(0)), res_3(\texttt{TMWrite(ok)}), inv_3(\texttt{TMEnd}), res_3(\texttt{TMEnd(commit)})\rangle$$

which is visualised as follows



□

Notation We use the following notation on histories: for a history h, $h \restriction p$ is the projection onto the events of process p only and $h[i..j]$ the subsequence of h from $h(i)$ to $h(j)$ inclusive. For a response event e, we let $rval(e)$ denote the value returned by e; for instance $rval(\texttt{TMBegin(ok)}) = \texttt{ok}$. If e is not a response event, then we let $rval(e) = \bot$. □

We are interested in three different types of histories. At the concrete level the TML implementation produces histories where the events are interleaved. h_1 above is an example of such a history. At the abstract level we are interested in *sequential histories*, which are ones where there is no interleaving at any level-transactions are atomic: completed transactions end before the next transaction starts. As part of the proof of opacity we use an intermediate specification which has *alternating histories*, in which transactions may be interleaved but operations (e.g., reads, writes) are non-interleaved.

A history h is *alternating* if $h = \langle\rangle$ or h is an alternating sequence of invocation and matching response events starting with an invocation. For the rest of this paper, we assume each process invokes at most one operation at a time and hence assume that $h \restriction p$ is alternating for any history h and process p. Note that this does not necessarily mean h is alternating itself. Opacity is defined for well-formed histories, which formalises the allowable

interaction between an STM implementation and its clients. Given a projection $h \restriction p$ of a history h onto a process p, a consecutive subsequence $t = \langle s_0, \ldots, s_m \rangle$ of $h \restriction p$ is a *transaction* of process p if $s_0 = inv_p(\texttt{TMBegin})$ and

- either $rval(s_m) \in \{\texttt{commit}, \texttt{abort}\}$ or s_m is the last event of process p in $h \restriction p$, and
- for all $0 < i < m$, event s_i is not a transaction invocation, i.e., $s_i \neq inv_p(\texttt{TMBegin})$ and not a transaction completion, i.e., $rval(s_i) \notin \{\texttt{commit}, \texttt{abort}\}$.

Furthermore, t is *committing* whenever $rval(s_m) = \texttt{commit}$ and *aborting* whenever $rval(s_m) = \texttt{abort}$. In these cases, the transaction t is *completed*, otherwise t is *live*. A history is *well-formed* if it consists of transactions only and there is at most one live transaction per process.

Example 2.2 The history h_1 given above is well-formed, and contains a committing transaction of process 3 and a live transaction of process 2. □

2.3. Opacity

The basic principle behind the definition of opacity (and similar definitions) is the comparison of a given concurrent history against a sequential one. Within the concurrent history in question, we distinguish between *live* and *completed* transactions; completed transactions are in turn split into committed (those that end by successfully committing) and aborted transactions (those that end by aborting). Opacity imposes a number of constraints, that can be categorised into three main types as follows:

1. There are two *ordering constraints*, which describe how events occuring in a concurrent history may be sequentialised, i.e., reordered to form a sequential history (with no interleaving of transactions).
   (a) Transactions must preserve program order [Lam79], i.e., the *order of operations within a process* must be maintained when sequentialising a concurrent history.
   (b) Like linearizability [HW90], transactions must obey a *real-time ordering* constraint, i.e., only transactions that overlap may be reordered when sequentialising a concurrent history. The real-time ordering constraint for opacity is weaker than linearizability in that operations that do not overlap may be reordered (in a different order to their real-time order) provided the transactions they belong to overlap.
2. There are four *semantic constraints* that describe validity of a sequential history hs obtained from a concurrent history (in a manner satisfying the ordering constraints above). A transaction may write to a location, then subsequently read from the same location; we refer to such a read as a *self-referencing* read.
   (a) Committing transactions in hs must be *serialised*, i.e., each write must modify memory in an appropriate manner, and each non self-referencing read must be consistent with all previous committed writes in hs.
   (b) Aborting transactions in hs must not modify memory (i.e., their writes must not take effect), and furthermore each non self-referencing read must be consistent with all previous committed writes in hs.
   (c) Live transactions in hs must behave as aborting transactions because it is not possible to determine whether or not a transaction will commit successfuly.
   (d) For all transactions (including live and aborting transactions) all self-referencing reads must be *internally consistent*, i.e., each self-referencing read to a location l must be consistent with the last previous write to location l by the same transaction.
3. There is a *prefix-closure constraint*, which requires that each prefix of a concurrent history can be sequentialised so that the ordering and semantic constraints above are satisfied.

There are numerous formalisations of opacity in the literature. Our presentation mainly follows Attiya et al. [AGHR13], but we explicitly include the prefix-closure constraint to ensure consistency with other accepted definitions [LLM12b, GK10, HLR10]. Without this prefix-closure constraint it is possible for a history to suffer from an "early read" anomaly, where a transaction may appear to see a committing write from a different transaction before the write physically takes place (see [GK10] for details).[4]

[4] Our original paper [DDS+15] presents the definition by Attiya et al. [AGHR13], without the prefix-closure property. However, the set of histories that we consider [DDS+15] are inductively generated by TML, and hence, ensures prefix-closure. Moreover, the two versions coincide whenever a prefix-closed set of histories is considered.

The most important difference between our formulation of opacity and the standard formulations to be found in [LLM12b, GK10, HLR10] is that in our setting, events are indexed by processes rather than transaction identifiers. This difference is motivated by a desire to be faithful to an STM interface that one would expect to find in a multithreaded library or programming language, where each process can execute several transactions. We discuss the consequences of this decision in Sect. 5.

We now formalise opacity via a series of definitions leading up to the definition of an opaque history (Definition 2.5). As we mentioned above, the basic principle is the comparison of a given concurrent history against a sequential one. The matching sequential history has to (a) consist of the same events, and (b) preserve the real-time order of transactions, and (c) events within processes need to have the same ordering. To help formalise this we introduce the following notation. We say a history h is *equivalent* to a history h' (capturing conditions (a) and (c)), denoted $h \equiv h'$, if for all processes $p \in P$, $h \restriction p = h' \restriction p$. Further, the *real-time order* on transactions t_1 and t_2 in a history h is defined as $t_1 \prec_h t_2$ if t_1 is a completed transaction and the last event of t_1 in h occurs before the first event of t_2. However, we do not consider arbitrary sequential histories as candidates for this matching. Rather, these need to satisfy the semantics constraints explained above. We detail one after the other in the following.

Sequential history semantics We now define formally the notion of sequentiality, noting that sequentiality refers to transactions: a sequential history is alternating and does not interleave events of different transactions. We first define non-interleaved histories.

Definition 2.1 *(Non-interleaved history)* A well-formed history h is *non-interleaved* if transactions of different processes do not overlap. That is, for any processes p and q and histories h_1, h_2 and h_3, if $h = h_1 \frown \langle inv_p(\texttt{TMBegin}) \rangle \frown h_2 \frown \langle inv_q(\texttt{TMBegin}) \rangle \frown h_3$ (where $\frown$ concatenates sequences) and h_2 contains no `TMBegin` operations, then either h_2 contains a response event e such that $rval(e) \in \{\texttt{abort}, \texttt{ok}\}$, or h_3 contains no operations of process p. □

In addition to being non-interleaved, a sequential history has to ensure that the behaviour is meaningful with respect to the reads and writes of the transactions. For this, we look at each address in isolation and define what a valid sequential behaviour on a single address is.

Definition 2.2 *(Valid history)* Let $h = \langle ev_0, \ldots, ev_{2n-1} \rangle$ be a sequence of alternating invocation and matching response events (cf. Table 1) starting with an invocation and ending with a response.

We say h is *valid* if there exists a sequence of states $\sigma_0, \ldots, \sigma_n$ such that $\sigma_0(l) = 0$ for all $l \in L$, and for all i such that $0 \leq i < n$ and $p \in P$:

1. if $ev_{2i} = inv_p(\texttt{TMWrite}(l, v))$ and $ev_{2i+1} = res_p(\texttt{TMWrite}(\texttt{ok}))$ then $\sigma_{i+1} = \sigma_i[l := v]$
2. if $ev_{2i} = inv_p(\texttt{TMRead}(l))$ and $ev_{2i+1} = res_p(\texttt{TMRead}(v))$ then $\sigma_i(l) = v$ and $\sigma_{i+1} = \sigma_i$.
3. for all other pairs of events (reads and writes with an abort response, as well as begins and ends) $\sigma_{i+1} = \sigma_i$.

We write $[\![h]\!](\sigma)$ if σ is a sequence of states that makes h valid (since the sequence is unique, if it exists, it can be viewed as the semantics of h). □

The point of STMs is that the effect of the writes only takes place if the transaction commits. Writes in a transaction that abort do not affect the memory. However, all reads must be consistent with previously committed writes. Therefore, only some histories of an object reflect ones that could be produced by an STM. We call these the *legal* histories, and they are defined as follows.

Definition 2.3 *(Legal histories)* Let hs be a non-interleaved history and i an index of hs. Let hs' be the projection of $hs[0..(i-1)]$ onto all events of committed transactions plus the events of the transaction to which $hs(i)$ belongs. Then we say hs *is legal at* i whenever hs' is valid. We say hs *is legal* iff it is legal at each index i. □

This allows us to define sequentiality for a single history, which we lift to the level of specifications.

Definition 2.4 *(Sequential history)* A well-formed history hs is *sequential* if it is non-interleaved and legal. We denote by $\mathcal{S}$ the set of all possible well-formed sequential histories. □

Opaque histories A given history may be *incomplete*, i.e., it may contain pending operations, represented by invocations that do not have matching responses. Some of these pending operations may be commit operations, and some of these commit operations may have taken effect: that is, the write operations of a committing transaction may be visible to other transactions. To account for this possibility, we define a set $complete(h)$ that contains all histories constructed by adding to h, $\texttt{TMEnd}(\texttt{commit})$ responses for any subset of the pending commit operations,

and adding TMEnd(abort) responses for all other pending operations. The sequential history must then have the same events as those of one of the histories in $complete(h)$. The successfully completed transactions may be visible to other transactions, and the aborted transactions must be invisible.[5]

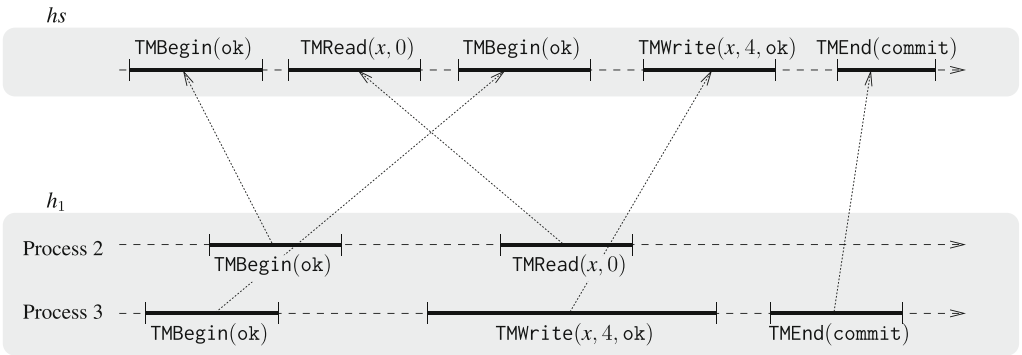
Definition 2.5 *(Opaque history)* A history h is *end-to-end opaque* iff for some $hc \in complete(h)$, there exists a sequential history $hs \in \mathcal{S}$ such that $hc \equiv hs$ and $\prec_{hc} \subseteq \prec_{hs}$. A history h is *opaque* iff each prefix h' of h is end-to-end opaque; a set of histories $\mathcal{H}$ is *opaque* iff each $h \in \mathcal{H}$ is opaque; and an STM implementation is *opaque* iff its set of histories is opaque. □

In Definition 2.5, conditions $hc \equiv hs$ and $\prec_{hc} \subseteq \prec_{hs}$ establish the ordering constraints and the requirement that $hs \in \mathcal{S}$ ensures the memory semantics constraints. Finally, the prefix-closure constraints are ensured because end-to-end opacity is checked for each prefix of h.

Example 2.3 The history h_1 in Example 2.1 is opaque; the corresponding sequential history is

$$hs \mathrel{\widehat{=}} \langle inv_2(\texttt{TMBegin}), res_2(\texttt{TMBegin(ok)}), inv_2(\texttt{TMRead}(x)), res_2(\texttt{TMRead}(0)), inv_3(\texttt{TMBegin}), res_3(\texttt{TMBegin(ok)}), inv_3(\texttt{TMWrite}(x, 4)), res_3(\texttt{TMWrite(ok)}), inv_3(\texttt{TMEnd}), res_3(\texttt{TMEnd(commit)})\rangle$$

The mapping from h_1 to hs is visualised as follows.



Reordering of TMRead(x, 0) and TMBegin(ok) in h_1 is allowed because their corresponding transactions overlap (even though the operations themselves do not). □

It is also instructive to consider histories that are not opaque. For simplicity, we only consider examples in which interleaving occurs at the level of transactions. More complex examples of histories that do not satisfy opacity where interleaving additionally occurs at the level of operations can also be constructed.

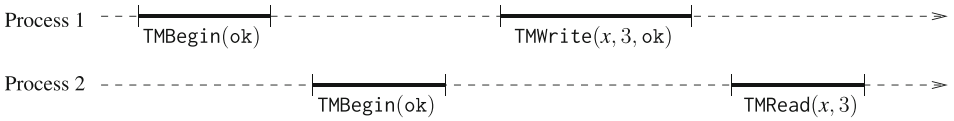
Example 2.4 A very simple example is h_2, which violates memory semantics, since it reads a value 4, that has not been written:

$$h_2 \mathrel{\widehat{=}} \langle inv_1(\texttt{TMBegin}), res_1(\texttt{TMBegin(ok)}), inv_1(\texttt{TMRead}(x)), res_1(\texttt{TMRead}(4))\rangle$$

A second more complex example is h_3 below

$$h_3 \mathrel{\widehat{=}} \langle inv_1(\texttt{TMBegin}), res_1(\texttt{TMBegin(ok)}), inv_2(\texttt{TMBegin}), res_2(\texttt{TMBegin(ok)}), inv_1(\texttt{TMWrite}(x, 3)), res_1(\texttt{TMWrite(ok)}), inv_2(\texttt{TMRead}(x)), res_2(\texttt{TMRead}(3))\rangle$$

which is visualised as follows



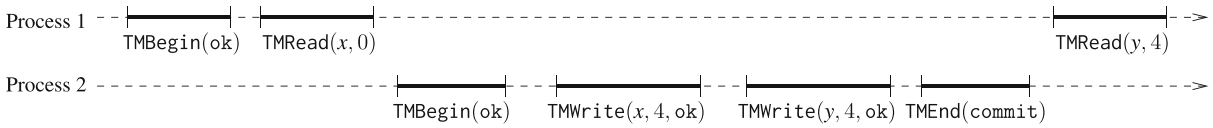
Transaction 2 reads value 3 written by transaction 1, which is still live. This is disallowed by opacity, since all values read must be from a state where only the effects of transactions that have already committed are visible.

[5] This process of completing transactions is similar to the completion of operations in the formalisation of linearizability [HW90].

Now consider a third example:

$h_4 \mathrel{\widehat{=}} \langle inv_1(\texttt{TMBegin}), res_1(\texttt{TMBegin(ok)}), inv_1(\texttt{TMRead}(x)), res_1(\texttt{TMRead}(0)),$
$inv_2(\texttt{TMBegin}), res_2(\texttt{TMBegin(ok)}), inv_2(\texttt{TMWrite}(x, 4)), res_2(\texttt{TMWrite(ok)}),$
$inv_2(\texttt{TMWrite}(y, 4)), res_2(\texttt{TMWrite(ok)}), inv_2(\texttt{TMEnd}), res_2(\texttt{TMEnd(commit)}),$
$inv_1(\texttt{TMRead}(y)), res_1(\texttt{TMRead}(4))\rangle$

which is visualised as follows:



In h_4 transaction 1 reads $x = 0$ from initial memory, then transaction 2 runs, which writes $x = y = 4$ and commits. Finally transaction 1 reads $y = 4$. This also violates opacity, since it is not possible to order the transactions sequentially: either transaction 1 is ordered first (and reads $x = y = 0$), or transaction 2 is ordered first (in which case transaction 1 should read $x = y = 4$). In general, however, an implementation could allow process 1 to read $y = 0$, i.e., if we replace the last event in h_4 by $res_2(\texttt{TMRead}(0))$, the modified history satisfies opacity.□

Our TML example cannot generate histories h_2, h_3, or h_4. History h_2 is avoided because all successful reads and writes take place directly in memory. For both h_3 and h_4, the final read is not possible—in both histories, these reads will abort because they will detect that the write in a different process has incremented `glb`.

This example also illustrates the relationship with strict serializability. For example, neither h_3 nor h_4 violate strict serializability [Pap79]. To satisfy strict serializability, for h_3 we must guarantee that transaction 1 always commits, while for h_4 we require that transaction 1 detects the inconsistent reads when attempting a commit, and to abort.

Strict serializability is too weak, and histories such as h_4 are problematic for implementations in which reads and writes to transaction variables are alternated with computations that use these values. To see this, suppose all committing transactions are required to preserve the invariant $x = y$ (the transactions in h_4 satisfy this invariant). Then, assuming all transactions act as if they are atomic, transaction 1 could rely on reading equal values for x and y. Even though transaction 1 will not be able to successfully commit, it could attempt to compute $x/(y+4-x)$ after reading x and y, which would give an unexpected division by zero.

Returning to our question of implementation correctness of the TML, this can now be rephrased as: *Are all the well-formed histories generated by TML opaque?* Having provided the necessary formalism to pose this question, we now explain our first proof method for showing opacity of TML.

3. Proving opacity: method 1—using linearizability

Proving opacity of an STM object is difficult, as it determines a relationship between a fine-grained implementation in which individual statements (and hence, operations) may be interleaved, in terms of a sequential specification in which sequences of transactional memory operations are considered atomic.

To bridge the large gap between the implementation level in which there are interleavings on both the level of operations and transactions, and the sequential specification in which neither operations nor transactions are interleaved, we split the proof into two steps. We first show that in TML all operations (i.e., write, read, start, begin) act as though they occur *atomically* at one point in between its invocation and response: we show TML to be *linearizable* [HW90]. This allows us to carry out the proof of opacity of TML in a simpler setting: we only prove opacity for the *alternating* histories of TML, i.e., the ones in which every invocation is directly followed by the response. Proving opacity for an alternating history is simplified by the fact that it is easy to define the semantics of read and write operations over these alternating histories (Sect. 3.4). Specifying these semantics directly over histories with interleaved operations would be much more difficult. Figure 2 visualises the overall approach.

Our proofs have been automated in KIV [EPS+14], the resulting development may be viewed online [TML16]. The link also contains additional notes on our KIV proof.

This section is organised as follows. Section 3.1 formalises linearizability and describes how it can be used to prove opacity. Section 3.2 presents the KIV model, and the two proof steps in the KIV proof are given in Sects. 3.3 and 3.4.

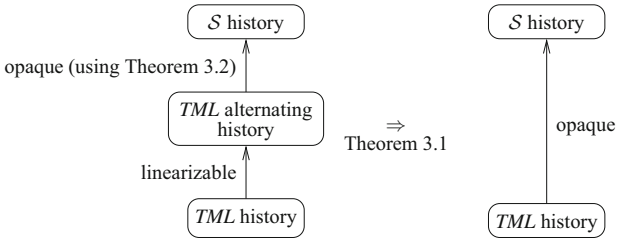


Fig. 2. Method 1: proof steps

3.1. Linearizability and opacity

The first part of the correctness proof of the TML implementation proceeds by showing it to be *linearizable* [HW90], which is the standard correctness criterion for concurrent objects. Linearizability provides the illusion that each operation applied by concurrent processes takes effect instantaneously at some point between its invocation and its return. This point is known as the *linearization point*. In other words, if two operations overlap, then their output agrees with the sequential execution determined by the order of linearization points.

As with opacity, the formal definition of linearizability is given in terms of histories (of invocation/response events); for every concurrent history an equivalent alternating (invocations immediately followed by the matching response) history must exist that preserves real time order of operations. The *real-time order* on operation calls[6] o_1 and o_2 in a history h is defined as $o_1 \prec\!\!\!\prec_h o_2$ if the return of o_1 precedes the invocation of o_2 in h.

Linearizability differs from opacity in that it does not deal with transactions; thus transactions may still be interleaved in a matched alternating history. As with opacity, the given concurrent history may be incomplete. Thus the definition of linearizability uses a function *complete* that adds matching returns to some of the pending invocations to a history h, and then removes any remaining pending invocations.

Definition 3.1 *(Linearizability)* A history h is *linearized* by alternating history ha, if there exists a history $hc \in complete(h)$ such that $hc \equiv ha$ and $\prec\!\!\!\prec_{hc} \subseteq \prec\!\!\!\prec_{ha}$. A concurrent object is linearizable with respect to a specification $\mathcal{A}$ (a set of alternating histories) if for each concurrent history h, there is an alternating history $ha \in \mathcal{A}$ that linearizes it. □

With linearizability formalised, we now present the main theorem for our proof method, which enables opacity to be proved via histories of the atomic specification of an STM.

Theorem 3.1 A concrete history h is opaque if there exists an alternating history ha such that h is linearizable with respect to ha and ha is opaque.

Proof Suppose (a) h is linearizable with respect to ha and (b) ha is opaque with respect to hs. Then, by (a), there exists a history $hc \in complete(h)$ such that $hc \equiv ha$ and $\prec\!\!\!\prec_{hc} \subseteq \prec\!\!\!\prec_{ha}$ and by (b), there exists a well-formed sequential history hs such that $ha \equiv hs$ and $\prec_{ha} \subseteq \prec_{hs}$. We must show that $hc \equiv hs$ and $\prec_{hc} \subseteq \prec_{hs}$ holds. Clearly, $hc \equiv hs$ because $\equiv$ is transitive, and if $\prec\!\!\!\prec_{hc} \subseteq \prec\!\!\!\prec_{ha}$ and $\prec_{ha} \subseteq \prec_{hs}$, then $\prec_{hc} \subseteq \prec_{hs}$ because preserving the real-time order of operations also preserves the real-time order of transactions. □

Linearizability proofs often use a set $\mathcal{A}$ of alternating histories which is generated by abstract operations that use a different state space than the concrete algorithms, e.g. the abstract operations may work on a queue, while the concrete algorithms manipulate pointer structures. In applying our result to TML, we do not change the state space, since the final set $\mathcal{S}$ of sequential histories we have to use in the opacity proof are histories of the same algorithm. Therefore we prove that every concurrent history h will be linearized by an alternating history $ha \in \mathcal{A}$, where $\mathcal{A}$ is the set of alternating histories that the original algorithm may produce by sequentially executing operations.

The alternating histories ha considered in linearizability are all complete, i.e. each invocation is immediately followed by a corresponding return. Therefore reasoning over alternating histories can be simplified by abstracting each invocation/response pairs into a single *run event* as summarised in Table 2. Run event $Begin(p)$ denotes a `TMBegin` operation by process p; run events $Read(p, l, v)$ and $Write(p,l,v)$ denote successful read and write operations by process p on address l with value v; run event $Commit(p)$ denotes a successful `TMEnd` operation by process p; and $Abort(p)$ denotes an operation invocation that aborts.

[6] Note: this is different from the real time order on transactions defined in Sect. 2.3

Table 2. Run events abstracting matching invocation/return pairs

run events	possible sequential invocation/response pairs
$Begin(p)$	$\langle inv_p(\texttt{TMBegin}), res_p(\texttt{TMBegin(ok)})\rangle$
$Read(p, l, v)$	$\langle inv_p(\texttt{TMRead}(l)), res_p(\texttt{TMRead}(v))\rangle$
$Write(p, l, v)$	$\langle inv_p(\texttt{TMWrite}(l, v)), res_p(\texttt{TMWrite(ok)})\rangle$
$Commit(p)$	$\langle inv_p(\texttt{TMEnd}), res_p(\texttt{TMEnd(commit)})\rangle$
$Abort(p)$	$\langle inv_p(\texttt{TMRead}(l)), res_p(\texttt{TMRead(abort)})\rangle$, $\langle inv_p(\texttt{TMWrite}(l, v)), res_p(\texttt{TMWrite(abort)})\rangle$, $\langle inv_p(\texttt{TMEnd}), res_p(\texttt{TMEnd(abort)})\rangle$

A *run* is a sequence of run events. Thus we specifically show that the run r corresponding to ha is opaque.

Example 3.1 The run corresponding to the history

$ha \mathrel{\widehat{=}} \langle inv_2(\texttt{TMBegin}), res_2(\texttt{TMBegin(ok)}), inv_2(\texttt{TMRead}(l), res_2(\texttt{TMRead(0)}), inv_3(\texttt{TMBegin}), res_3(\texttt{TMBegin(ok)}),$
$inv_3(\texttt{TMWrite}(l, 4)), res_3(\texttt{TMWrite(ok)}), inv_3(\texttt{TMEnd}), res_3(\texttt{TMEnd(commit)}),$
$inv_2(\texttt{TMRead}(l)), res_2(\texttt{TMRead(abort)})\rangle$

is $\langle Begin(2), Read(2, l, 0), Begin(3), Write(3, l, 4), Commit(3), Abort(2)\rangle$. □

Because `Abort(p)` relates to several possible pairs of events, a run is more abstract than a complete alternating history. However from the view of opacity, the pairs that `Abort(p)` represents are indistinguishable, so in effect the encoding just simplifies the mechanized proof.

3.2. Modelling TML in KIV

Before we discuss the proof steps, we first describe how the different specifications are modelled in KIV.

The concrete specification: To model the concrete state of the TML, we use KIV's record type, which is used to define a constructor `mkcs` (make concrete state *cs*) containing a list of fields of some type. Field `glb` represents the global variable *glb*, and `mem` represents the memory state and hence maps addresses to values (in this case integers). Local variables are mappings from processes (of type `Proc`) to values; for the TML, we have local variable `pc` for the program counter, `loc` for the local copy of `glb`, as well as variables `l` (representing `addr` in the pseudocode) and `v` (representing `val` in the pseudocode). We thus use the following state.

```
CState =
mkcs(. .glb : nat, . .mem : L → V, . .pc : Proc → PC,
     . .loc : Proc → nat, . .l : Proc → L, . .v : Proc → V)
```

Modelling atomic statements: Modelling one step of a TML algorithm as a KIV state transition is done in two stages. We first model the step of one process working on global variables `glb` and `mem` and its local variables `pc`, `loc`, `l` and `v`. As an example, consider the statement labelled `W2`, which is modelled by `write2-def` below. Here, `COP` is used to denote that the step is internal (i.e., neither an invocation nor a response; such steps have an additional input resp. output parameter) and `write2` is the index of the operation. Modifications to `glb` and `pc` are conditional, denoted by ⊃, on the test `loc = glb`. Thus, if `loc = glb`, then `pc'` is set to `W3`, otherwise `pc'` is set to `W6`. The transitions alter the concrete state, the after state is denoted by dashed variables.

```
write2-def:
⊢ COP(write2)(glb, mem, pc, loc, l, v, glb', mem', pc', loc', l', v')
↔
( pc = W2 ∧ loc' = loc ∧ mem' = mem ∧ l' = l ∧ v' = v
     ∧ glb' = (loc = glb ⊃ loc + 1;glb) ∧ pc' = (loc = glb ⊃ W3;W6) );
```

Two special `pc` values `N` and `T` are used to enforce well-formed executions of transactions: `pc = N` is the inital value, indicating that the process is currently not executing a transaction. Only the invoking step of `TMBegin` is enabled in this case. The aborting steps (starting at `pc = R4, W3`) of `TMRead` and `TMWrite` as well as the returning

step of TMEnd set pc' = N. When pc = T, the process is within a transaction, but currently not executing an operation. Successful return steps of reads and writes (starting with pc = R3, W6) set pc' = N. The invokung steps of TMRead, TMWrite and TMEnd are enabled when pc = T.

Promotion to system wide steps The process-local specification of a step is then promoted to a step COp(cj,p) done by process p on the full state cs. The promotion uses function update, where $f[p := d]$ is function f with p overwritten to return d.

The interesting part of this promotion is that we add an auxiliary variable r that records run events, which describes the intermediate level of our proof. The variable r is updated exactly at the *linearization points* of each of the operations TMBegin, TMRead, TMEnd, and TMWrite.

The linearization points of each operation are annotated in comments in the code in Fig. 1. The expression on the right of "r' =" below is an if-then-else expression describing r', i.e. the value of r in the post state. Thus, for example, the first line states that r' is set to r + Begin(p), which concatenates Begin(p) to r, whenever the instruction at B1 is executed and an even value of glb is loaded into loc. In this case, the operation will definitely go on to start a transaction as the outcome of the next test at B2 is determined locally.

Operation TMRead linearizes at R2 to a non-aborting Read if the value of glb is the same as the stored value in loc, and linearizes to an aborting Read if the value of glb changes. Operation TMWrite linearizes successfully when the memory is updated at W5, and linearizes to Abort if the cas at W2 fails. Finally, operation TMEnd never aborts, yet there are two linearization points depending on whether the transaction has successfully executed a TMWrite. If no writes were performed, then loc must be even; in such a transaction TMEnd must linearize at E1. Otherwise if the transaction had performed a successful write, then loc must have been set to an odd value at W4, therefore, the linearization point for TMEnd for such a transaction is E2. The last line of the formula covers all remaining cases, where no LP is executed, and r' is kept equal to r.

```
   COp(cj, p)(cs, r, cs', r')
↔ ∃ pc', loc', l', v'.
          COP(cj)(cs.glb, cs.mem, cs.pc(p), cs.loc(p),cs.l(p), cs.v(p),
                  cs'.glb, cs'.mem, pc', loc', l', v')
        ∧ cs'.pc = cs.pc[p := pc'] ∧ cs'.loc = cs.loc[p := loc']
        ∧ cs'.l = cs.l[p := l'] ∧ cs'.v = cs.v[p := v']
        ∧ r' = (pc = B1 ∧ even(glb) ⊃ r + Begin(p) ;
                (pc = R2 ∧ loc = glb ⊃ r + Read(p, l, v) ;
                (pc = R2 ∧ loc ≠ glb ⊃ r + Abort(p) ;
                (pc = W2 ∧ glb ≠ loc ⊃ r + Abort(p) ;
                (pc = W5 ⊃ r + Write(p, l, v) ;
                (pc = E1 ∧ even(loc) ⊃ r + Commit(p) ;
                (pc = E2 ⊃ r + Commit(p) ;
                r ))))))));
```

3.3. Step 1: proving linearizability with respect to the intermediate specification

Having described how we model the TML implementation in KIV, including the embedding of the linearization points in the promoted operations, the next step is to show that every history h of this TML implementation is linearized by an alternating history of the intermediate specification. To simplify the proof, the alternating histories have been represented by runs and the run we want to use is already included as an auxiliary variable in the code.

Proving that each history h created by interleaved run is linearized to an alternating history represented by run r is done by proving two lemmas in KIV for each of the four operations (TMWrite etc).

The first lemma states: When executed by process p, no operation ever passes more than one linearization point (LP) in any execution (regardless whether other processes executing instructions interleaved with the steps of p) before executing a return (so in particular, even if TMBegins gets stuck in its loop it never executes more than one LP).

The second lemma states: If an operation reaches a return and terminates, then the operation has executed exactly one LP, i.e. exactly one run event of process p has been added to the run r. The arguments of this run event agree with the actual input/output of the invoking/response transition. As an example, the write operation adds $Write(p, l, v)$ to r when executing the instruction at W5 (and therefore actually writes v to $mem(l)$), and we prove that this is possible only when the input to the invoking instruction of TMWrite is l, v and the output is empty.

That these two lemmas are sufficient for proving linearizability, i.e. that any interleaved history h will linearize to r can be argued as follows. The second lemma implies that for any finished operation, where h contains a matching pair, the run r will contain a unique run event that represents this matching pair. For any operation that is still executing, h will have an invoke event, and there are two cases for r: either it will contain a run event that represents this invoke event and some (yet to be executed) return event, or no event (according to the first lemma). Thus in the definition of linearization, $complete(h)$ can be chosen to add the return events for all operations, where the first case applies, i.e., where the operation has passed its LP, and drop the pending invoke otherwise. This results in a history $complete(h)$ with the same matching pairs as those represented by r with a one-to-one mapping between them. The mapping preserves real-time order (as required by linearizability) since if one operation finishes before another one starts, the former will definitely execute its LP earlier, so the corresponding run event in r will be earlier in the sequence than the one of the second operation.

The proofs for linearizability with this encoding are rather simple. We exploit here that in our example the LPs of an operation executed by process p can be placed at the execution of an instruction by *this* process, and do not depend on future executions (more intricate examples where linearization points are executed by other threads or depend on the future run require more complex techniques [SDW14, DD15]). The method used here is akin to the technique used by Vafeiadis [Vaf07], where concrete states are augmented with auxiliary variables representing the abstract state together with additional modifications of the auxiliary state at the linearization points. Since we have chosen the LP of `TMWRite` to be the actual write to memory, abstract and concrete memory never differ, so we can even avoid an auxiliary variable for abstract memory.

3.4. Step 2: proving opacity of alternating histories using runs

In this subsection we prove an alternating history which linearizes a concurrent TML history is itself opaque. Together with the results defined above this will be sufficient to show opacity of the TML.

Firstly, we define opacity for runs, and show that proving opacity of runs is equivalent to proving opacity of alternating histories. Secondly, we discuss the KIV proof of opacity for TML runs. (Note that the descriptions below differ slightly from the actual KIV proof online; as we use modified function names here to keep this paper self-contained, i.e., the proof can be understood without having to refer to the KIV specification online.)

Defining opacity for runs Many of the definitions follow over from the definitions for histories in Sect. 2. We also need to define the semantics of a valid run on a sequence of states. To define opacity of a run, we first define the semantics of each run event from Table 2 on the memory state $mem \in Mem$ to produce the next state mem'. Notation $mem[l := v]$ denotes functional override, where $mem(l)$ is updated to v.

$$
\begin{aligned}
[\![Begin(p)]\!](mem, mem') &\mathrel{\widehat{=}} mem' = mem \\
[\![Read(p, l, v)]\!](mem, mem') &\mathrel{\widehat{=}} mem' = mem \wedge mem(l) = v \\
[\![Write(p, l, v)]\!](mem, mem') &\mathrel{\widehat{=}} mem' = mem[l := v] \\
[\![Commit(p)]\!](mem, mem') &\mathrel{\widehat{=}} mem' = mem \\
[\![Abort(p)]\!](mem, mem') &\mathrel{\widehat{=}} mem' = mem
\end{aligned}
$$

Since TML is an eager algorithm, we do not need to distinguish the semantics of run events used in defining the set $\mathcal{A}$ of legal alternating histories in Definition. 3.1 from the semantics of run events viewed as abbreviations of two events needed for opacity. (cf. Definition 2.2 of a valid history). The latter definition is always as above, the former definition has to specify the effect of sequentially executing the operations of TML on memory. Here, both are the same. For a lazy algorithm that collects a write-set in its write operation, the semantics of `Write(p)` would be identity, while the semantics of `Commit(p)` would apply the write-set to get from mem to mem'.

The semantics of individual run events are lifted to the level of runs as follows. Below, σ is a sequence of memory states and $\#\sigma$ defines the length of σ, which by the first conjunct is one more than the length of `r`. By the second conjunct, for each n, the transition from $\sigma(n)$ to $\sigma(n + 1)$ is generated using $r(n)$. Because the memory state has been made explicit, $[\![r]\!](\sigma)$ only holds for valid and legal runs.

$$[\![r]\!](\sigma) \mathrel{\widehat{=}} \#\sigma = \#r + 1 \wedge \forall\, n \bullet n < \#r \Rightarrow [\![r(n)]\!](\sigma(n), \sigma(n + 1));$$

Finally, we define opaque runs as follows, where run r is mapped to sequential run rs. Predicate $r \equiv rs$ ensures equivalence between r and rs, predicate $\prec_r \subseteq \prec_{rs}$ ensures real-time ordering is preserved, and $\neg interleaved(rs)$ states that transactions in rs may not overlap. The final conjuct ensures rs is both valid and legal as defined

in Definitions 2.2 and 2.3, respectively, where *committed* restricts a given run to the run events of committed transactions plus the (live) transaction to which $r(n)$ belongs as defined in Definition 2.3.

$$ee\_opaque(r, rs) \mathrel{\widehat{=}} (r \equiv rs) \wedge (\prec_r \subseteq \prec_{rs}) \wedge \neg interleaved(rs) \wedge \\ \forall n \bullet n < \#rs \Rightarrow \exists \sigma \bullet \sigma(0) = (\lambda x \bullet 0) \wedge [\![committed(rs[0..n])]\!](\sigma)$$

We must now ensure that proving opacity of runs is sufficient for proving opacity of complete alternating histories. This is established via the following theorem. We say a run r *corresponds* to an alternating history ha iff r can be obtained from ha by replacing each pair of matching events in ha by the corresponding run event from Table 2.

Theorem 3.2 An alternating history ha is end-to-end opaque if there exists a run r that *corresponds* to ha and there is an rs such that $ee\_opaque(r, rs)$ holds.

Proof The proof of this theorem is straightforward as the definition of opacity of a run is built on the opacity of an alternating history. □

The invariants for opacity The rest of the proof is now about proving that for each state (cs, r) of the TML augmented with runs, it is possible to find a sequential run rs such that $opaque(r, rs)$ holds.

As with our work on linearizability [SDW14], we prove this via construction of an appropriate invariant. The main proof then shows that all augmented states (cs, r) satisfy the predicate $\exists rs \bullet INV(cs, r, rs)$. The formula $INV(cs, r, rs)$ defines a number of invariants that, in particular, imply $opaque(r, rs)$.

Informally, formula $INV(cs, r, rs)$ encodes the observation that the (legal) transaction sequences rs generated by the TML implementation always consist of three parts.

1. A first part that alternates finished transactions and live transactions with an even value for *loc(p)* that is already smaller than the current value of *glb*. The processes p executing such live transactions have only done reads. They are still able to successfully commit, but they are no longer able to successfully read or write.
2. A second part that consists of transactions of processes p that have *loc(p) = glb* (or *loc(p)+1 = glb*, in case a writing transaction exists). These are the live readers, that still are able to do more reads if there is no writer.
3. Finally, an optional live writing transaction. The process p executing this transaction either satisfies $odd(loc(p)) \wedge loc(p) = glb$ or $pc(p) = W4 \wedge odd(glb) \wedge glb = loc(p) + 1$.

That the partitioning is an invariant is established by proving some additional simpler properties of the TML implementation with respect to the corresponding sequential run rs. The most important ones are as follows, where p is assumed to be the process generating the transaction.

INV1 Transactions for which $loc(p)$ is even have not performed any writes.

INV2 Any live transaction with an odd value of $loc(p)$ is the last transaction in rs, and $loc(p) = glb$ in this case. As a direct consequence there is at most one live transaction with an odd value for $loc(p)$.

INV3 If the sequential run rs contains a live transaction t by process p with $loc(p) = glb$ or $pc(p) = W4$, any finished transaction must occur before t. This ensures that when a live transaction becomes a writer, it must not be reordered with a finished one, but only with live readers.

INV4 Live transactions are ordered (non-strictly) by their local values of loc. This property is crucial for preserving real-time order, since a larger loc implies that the transaction has started later.

INV5 Strengthening $opaque(r, rs)$, the state sequence σ that is needed to ensure that the last event of rs is valid (cf. Definition 2.3) always ends with current memory. Formally, for any augmented state cs, r the sequential history rs is such, that for its projection rs' to events of committed transactions plus the events of the last transaction a (unique) state sequence σ with $[\![rs']\!]\sigma$ exists where the last element of σ is equal to $cs.mem$.

INV6 Aborted transactions contain no write operations.

Opacity proof in KIV The proof proceeds by assuming $INV(cs, r, rs)$ holds for some rs, and shows that the invariant holds after any step of the TML specification. If the step generates cs', r' it must be possible to construct a new sequence rs' such that $INV(cs', r', rs')$ holds. For all steps that do not linearize (i.e. do not modify r) this is easy, we simply choose $rs' = rs$. Therefore, each of these proofs except for the operation at `W4` (that increments loc and therefore could violate one of the invariants defined above) is trivial.

To explain the proofs, note that by construction rs is always a sequence of transactions, i.e. it has the form $ts(0) \frown ts(1) \frown \cdots \frown ts(\#ts-1)$ concatenating the events of individual transactions, each executed by some process. We therefore can assume a function $tseq(rs)$, which generates this sequence of transactions from rs in order.[7]

This sequence satisfies the invariants above, so as already discussed, it consists of three parts: (1) Completed transactions and "old" live readers, (2) live readers, and (3) possibly one writer at the end. Each linearization step of a TML operation executed by process p adds a corresponding run event re to r, i.e. $r' = r \frown \langle re \rangle$, and we have to show that we can find an appropriate rs'. With the exception of starting a new transaction with $re = Begin(p)$ (which simply adds the same event at the end of rs) the new rs' then must add re to the end of the appropriate live transaction, say $ts(j)$, leaving all others unchanged. The sequence ts' then may reorder the transactions $ts(i)$ such that opacity (and in particular the memory semantics of rs) is preserved. In most cases there is no need to reorder, i.e. the choice for ts' is $ts[j := ts(j) \frown \langle re \rangle]$.

Because opacity holds for the transaction sequence rs before the LP step, we know from Definition 2.5 that for each transaction $ts(k)$ a memory sequence σ_k exists, that fits the run events of the committed transactions before k together with the run events in $ts(k)$. In the following, we refer to σ_k as *the memory sequence validating* $ts[0..k]$. The task then is to find for each k a new memory sequence σ'_k that validates $ts'[0..k]$. There are three cases.

1. For $k = j$, we choose $\sigma' := \sigma \frown \langle mem' \rangle$, where mem' is computed from the last element $mem := last(\sigma)$ by applying the semantics of the added event re on $last(\sigma)$. Note that this preserves property **INV5**, since in the case of a write operation property **INV2** guarantees (since loc is odd at `W5`) that $rs[k]$ in this case is the last transaction, and the memory update of the algorithm agrees with the one done by the operation.
2. For $k < j$, we choose $\sigma'_k = \sigma_k$, since the extended transaction is not present.
3. For $j < k$, we choose $\sigma'_k = \sigma_k$ when re is not a commit. The difficult case remaining is the one where $ts'(j)$ is committing. However, because $ts'(j)$ is not the last transaction in the sequence, it cannot have an odd loc due to **INV2**, and by **INV1**, the transaction has not performed any writes. Therefore, the memory sequence σ'_j that validates $ts'[0..j]$ is of the form $\sigma_0 \frown \langle mem \rangle^n$, where $\langle mem \rangle^n$ is a sequence of mems of length n. The memory sequence σ_k that validates $ts[0..k]$ has prefix $\sigma_0 \frown \langle mem \rangle^n$, since $j < k$. Therefore, $\sigma_k = \sigma_0 \frown \langle mem \rangle^n \frown \sigma'$ and the new memory sequence that validates $ts'(k)$ can be set to $\sigma_0 \frown \langle mem \rangle^{n+1} \frown \sigma'$.

This proves the main invariant that rs' is legal. However, there is an additional problem when (a) run event re is a $Commit$ or $Abort$ or (b) $loc(p)$ is incremented at `W4`. Case (a) may violate **INV3**, which is necessary to ensure that real-time order in r is preserved, (b) may violate **INV4**.

The simplest example is a run $r = \langle Begin(1), Begin(2) \rangle$. A possible transaction sequence is simply $ts = \langle t_1, t_2 \rangle$ where $t_1 = \langle Begin(1) \rangle$ and $t_2 = \langle Begin(2) \rangle$. For scenario (a), process 2 may linearize a `TMEnd` resulting in $r' = r \frown \langle Commit(2) \rangle$. The new transaction sequence ts' must now have the completed transaction 2 *before* the live transaction 1 due to **INV3**, i.e. $ts' = \langle t'_2, t_1 \rangle$ where $t'_2 = \langle Begin(2), Commit(2) \rangle$. For scenario (b), process 1 may linearize a `TMWrite`, i.e. $r' = r \frown \langle Commit(1) \rangle$. Transaction 1 must now be moved to the end of the transaction sequence to preserve **INV4**. This results in $ts' = \langle t_2, t'_1 \rangle$ where $t'_1 = \langle Begin(1), Write(1, l, v) \rangle$. Note that swapping t_1 and t_2 is possible in both cases, since both transactions have done reads only, implying that the memory sequence needed to validate both transaction sequences $\langle t_1, t_2 \rangle$ and $\langle t_2, t_1 \rangle$ repeats a constant memory.

In general, for both scenarios, a transaction with current $loc(p)$ value must be moved. Case (a) must move the committing reader to the start among those whose value of loc equals $loc(p)$. In terms of the split of the transaction sequence into three parts, the transaction was one of the transactions of part 2, and must now become the last transaction of part 1. Case (b) must move the transaction that executes `W4` to the end of ts (it moves from part 2 to become the single writer of part 3). Both cases can be reduced to a lemma, that says that adjacent transactions $ts(n)$, $ts(n+1)$ executed by processes p and q, respectively, can be reordered whenever $loc(p) = loc(q)$. This is because by property **INV2**, both $loc(p)$ and $loc(q)$ must be even and by **INV1** neither may have performed any writes.

[7] Technically, a transaction sequence ts is represented in KIV as a sequence of ranges $m_i..n_i$, such that m_i and n_i mark the first and last event of a transaction in r. Assuming $r[m_i] = Begin(p_i)$, the events of transaction $ts(i)$ then are specified as $ts(i) = r[m_i..n_i] \restriction p_i$. The opacity predicate is therefore defined directly in terms of the range sequence instead of using rs.

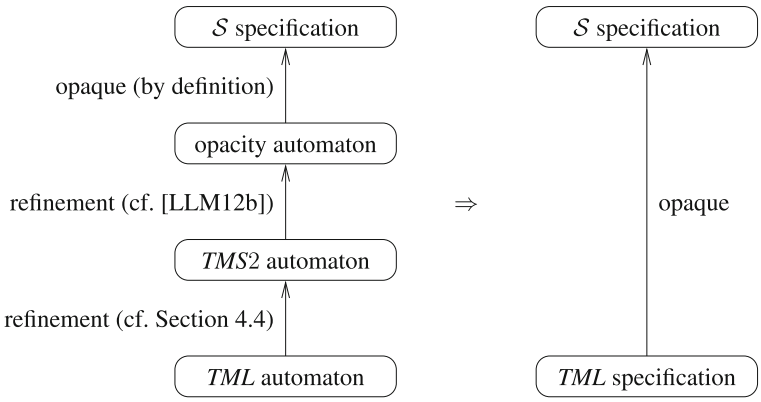


Fig. 3. Method 2: proof steps

Mechanization The entirety of Method 1 has been mechanized in KIV, excepting elements of the development that directly involve the notion of *histories* from Sect. 2. The definition of opacity over histories presented in Sect. 2.3 has not been mechanized. However, a definition of opacity over runs has been mechanized along with all the definitions on which it depends, and the definition of ee_opaque. Theorems 3.1 and 3.2 (which connect runs to histories) have not been mechanized. The definitions and proofs in the remainder of the development have been fully mechanized. In particular: the TML model; the linearizability proof outlined in Sect. 3.3; and the semantics and invariant proofs described in Sect. 3.4.

Specifying and proving opacity using KIV required four weeks of work. In particular, half the time was invested to develop an elegant formalisation of transactions that does not have to refer to auxiliary data like transaction identifiers and does not have to explicitly specify permutations. The most difficult part of the proof was figuring out a good lemma that gives criteria for preserving the semantics. This proof and the proofs of the main goals for each of the 7 linearization points and $pc(p) = W4$ are rather complex. They each have between 50 and 100 interactions. Our first guess for defining the invariant left out the two properties **INV4** and **INV6**, they were added during the proof, which also took ca. two person weeks. Streamlining these techniques in the context of a larger example (e.g., the TL2 algorithm [DSS06]) is a topic of future work.

4. Proving opacity: method 2—using a simulation with TMS2

The method in Sect. 3 achieves some level of decomposition for opacity proofs by linearizing TML histories to alternating histories. However, the proof method can be improved in two key areas: (1) The alternating history that we obtain must be coupled with the execution of the TML to prove opacity, i.e., the proof of opacity for the alternating histories relies on additional properties of the TML algorithm. (2) Using the alternating histories of the concrete algorithm represents a design choice, which gives a simple characterization of the effect on memory for eager algorithms only.

In this section, we address both issues and present an alternative proof of correctness of the TML algorithm, following a structure that is conventional for simulation proofs. This new proof leverages two existing results from the literature: the TMS2 specification by Doherty et al. [DGLM13], and the mechanized proof by Lesani et al. [LLM12b], which shows that the TMS2 specification implements an opaque specification (that generates all possible opaque histories). The specifications in [DGLM13, LLM12b] are formalised as Input/Ouput automata (IOA) [LT87], and hence, for the sake of continuity we also model the TML algorithm using IOA. An overview of this proof method is given in Fig. 3. All proofs between the TML and TMS2 specifications are fully mechanized using the Isabelle [NPW02] theorem prover.

This section is organised as follows. First we present some background (Sect. 4.1), in particular, the IOA formalism and its forward simulation proof rule. In Sects. 4.2 and 4.3 we present the TMS2 automaton (originally defined in [DGLM13]) and the TML automaton, respectively. The simulation proof as well as the invariants that we use are described in Sect. 4.4.

4.1. Input/output automata and forward simulation

We use Input/Output Automata (IOA) [LT87] to model both the specification and implementation.

Definition 4.1 An *Input/Output Automaton (IOA)* is a labeled transition system A with

- a set of states $states(A)$,
- a set of actions $acts(A)$,
- a set of start states $start(A) \subseteq states(A)$, and
- a transition relation $trans(A) \subseteq states(A) \times acts(A) \times states(A)$ (so that the actions label the transitions).

The set $acts(A)$ is partitioned into input actions $input(A)$, output actions $output(A)$ and internal actions $internal(A)$. The internal actions represent events of the system that are not visible to the external environment. The input and output actions are externally visible, representing the automaton's interactions with its environment. Thus, we define the set of *external actions*, $external(A) = input(A) \cup output(A)$. In the standard IOA setting, the distinction between input and output actions is important for composing automata, and reasoning about such compositions. However, the present work does not employ composition so we ignore this input/output distinction.[8]

An *execution* of an IOA A is a sequence σ of alternating states and actions, beginning with a state in $start(A)$, such that for all states σ_i except the last, $(\sigma_i, \sigma_{i+1}, \sigma_{i+2}) \in trans(A)$. A *reachable* state of A is a state appearing in an execution of A. An *invariant* of A is any superset of the reachable states of A (equivalently, any predicate satisfied by all reachable states of A). A *trace* of A is any sequence of (external) actions obtained by projecting the external actions of any execution of A. The set of traces of A represents A's externally visible behaviour. If every trace of an automaton C is also a trace of an automaton A, then we say that C *implements* A.

As highlighted in Fig. 3, we show that TML is correct by showing that the IOA specifying TML is a refinement of the IOA that specifies TMS2 [DGLM13]. One way of doing this is to prove the existence of a *forward simulation* from an implementation to the specification. The definition of forward simulation we use is adapted from that of Lynch and Vaandrager [LV95].

Definition 4.2 A *forward simulation* from a concrete IOA C to an abstract IOA A is a relation $R \subseteq states(C) \times states(A)$ such that each of the following holds.

Initialisation.
For each $cs \in start(C)$ there is some $as \in start(A)$ such that $R(cs, as)$.

External step correspondence.
For each $cs \in reach(C)$, $as \in reach(A)$, $a \in external(C)$ and $cs' \in states(C)$,
if $R(cs, as)$ and $(cs, a, cs') \in trans(C)$ then
there is some $as' \in states(A)$ such that $R(cs', as')$ and $(as, a, as') \in trans(A)$.

Internal step correspondence.
For each $cs \in reach(C)$, $as \in reach(A)$, $a \in internal(C)$ and $cs' \in states(C)$,
if $R(cs, as)$ and $(cs, a, cs') \in trans(C)$ then
either $R(cs', as)$ or
there is some $as' \in states(A)$ and $a' \in internal(A)$ such that $R(cs', as')$ and $(as, a', as') \in trans(A)$.

Forward simulation is *sound* in the sense that if there is a forward simulation between A and C, then C is a refinement of A [LV95, Mül98]. The notion of refinement here is trace inclusion, and hence, if there is a forward simulation between A and C, then every trace of C is also a trace of A.

Definition 4.2 is stronger (and less general) than the definition given by Lynch and Vaandrager [LV95] in two different ways, however, neither strengthening presents a problem for our mechanization.

1. In [LV95], an internal step of the concrete automaton can be simulated by several internal steps of the abstract automaton, whereas in Definition 4.2, each internal step corresponds to at most one abstract internal step. The level of generality in [LV95] is rarely required in practice, and it is simpler to work with the stronger step correspondence rule.
2. In [LV95] in both the internal and external step correspondence rules, reachability is only assumed of the concrete state, whereas Definition 4.2 assumes that both the abstract and concrete states are reachable. We occasionally use reachability of the abstract state, to ensure that certain partial function applications are properly defined, so again the stronger definition is simpler to use.

[8] The full IOA framework also includes features for reasoning about liveness. In this paper we are only interested in safety properties, so we do not describe these features.

Both IOA and forward simulation have been mechanized by Müller [Mül98], which is now part of the standard Isabelle distribution. We have mechanized a proof that our notion of forward simulation (Definition 4.2) is sound, i.e., implies trace refinement by showing that it implies the definitions of Müller [Mül98].

4.2. The TMS2 specification

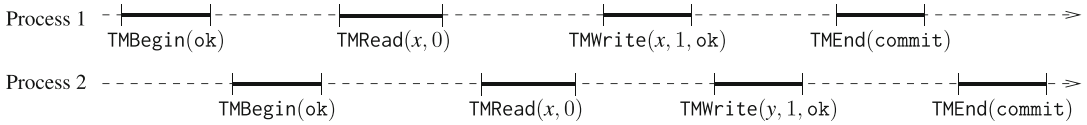
In this section we describe the TMS2 transactional memory specification [DGLM13], which is strictly stronger than opacity [LLM12b]. TMS2 is designed to capture structural patterns common to practical TM implementations, while being general enough to serve as a specification for a large class of implementations.

TMS2 imposes a stronger real-time ordering constraint on transactions than opacity, only allowing writing transactions to be reordered if their TMEnd operations overlap. Recall that opacity allows transactions to be reordered if they overlap regardless of whether or not their TMEnd operations overlap.

Example 4.1 To see this distinction, consider the history h_5 below.

$$h_5 \mathrel{\widehat{=}} \langle inv_1(\texttt{TMBegin}),\ res_1(\texttt{TMBegin(ok)}),\ inv_2(\texttt{TMBegin}),\ res_2(\texttt{TMBegin(ok)}),\ inv_1(\texttt{TMRead}(x)),\ res_1(\texttt{TMRead}(0)),$$
$$inv_2(\texttt{TMRead}(x)),\ res_2(\texttt{TMRead}(0)),\ inv_1(\texttt{TMWrite}(x, 1)),\ res_1(\texttt{TMWrite(ok)}),$$
$$inv_2(\texttt{TMWrite}(y, 1)),\ res_2(\texttt{TMWrite(ok)}),\ inv_1(\texttt{TMEnd}),\ res_1(\texttt{TMEnd(commit)}),$$
$$inv_2(\texttt{TMEnd}),\ res_2(\texttt{TMEnd(commit)})\rangle$$

which may be visualised as follows:



Because the TMEnd(commit) operations do not overlap, the ordering constraint for TMS2 requires that the transaction by process 1 be ordered before the transaction by process 2. However, doing so will result in a non-interleaved history that does not satisfy memory semantics. Opacity on the other hand permits the transaction by process 2 to be ordered first, which would result in a history consistent with a memory semantics, i.e., h_5 is opaque. □

Formally, TMS2 is specified by the IOA in Fig. 4, which describes the required ordering constraints, memory semantics and prefix properties. We assume a set T of *transaction identifiers* (*tids*), which are used to index actions and state variables. Also, we assume a set L of locations and a set V of values. Thus, memory is modelled by a function of type $L \to V$. A key feature of TMS2 is that it keeps track of a sequence of memory states, one for each committing transaction that contains a write. This makes it simpler to determine whether non self-referencing reads are consistent with previously committed write operations. Each committing transaction containing at least one write adds a new memory version to the end of the memory sequence.

The state space of TMS2 has several components. The first, $memories$ is the sequence of *memory* states. For each tid t there is a program counter variable pc_t, which ranges over a set of *program counter values* (called $PCVals$ in Fig. 4). Program counters are used to ensure that each transaction is well-formed, and to ensure that each transactional operation takes effect between its invocation and response. For each tid t there is also a a *begin index* variable $beginIdx_t$, that is set to the index of the most recent memory version when the transaction begins. This variable is critical to ensuring the real-time ordering property (see below). Finally, for each tid t there is a *read set*, $rdSet_t$, and a *write set*, $wrSet_t$, which record the values that the transaction has read and written during its execution, respectively. The read set is used to determine whether the values that have been read by the transaction are consistent with the same version of memory (see $validIdx$ below). The write set, on the other hand, is required because writes in $TMS2$ are modelled using *deferred update* semantics: writes are remembered in the transaction's write set, but are not published to any shared state until the transaction commits. Deferred update is very common among many transactional memory implementations, but is not used in TML. As we shall see, it is straightforward to account for this difference in the simulation relation.

Each action of TMS2 is indexed by a tid, and consists of a pair of functions 'Pre' and 'Eff' that describe the *precondition* and *effect* of the action on the state, respectively. For example, the action $resp_t$(TMBegin) can only be fired in a state satisfying pc_t = beginPending and its effect updates the value of pc_t to 'ready'. The external actions of TMS2 can be divided into two categories: *invocations* and *responses*, with invocations and responses for each transactional operation. TMS2 has internal actions for each read, write and commit operation, where the action name is prefixed by "do". These internal actions can be thought of as the points where the transactional operations "take effect", similar to the linearization points of the KIV proof in Sect. 3.

State variables:
$memories : \mathbb{N} \pfun L \to V$, initially satisfying $\mathrm{dom}\, memories = \{0\}$ and $initMem(memories(0))$
$pc_t : PCVal$, for each $t \in T$, initially $pc_t =$ notStarted for all $t \in T$
$beginIdx_t : \mathbb{N}$ for each $t \in T$, unconstrained initially
$rdSet_t : L \pfun V$, initially empty for all $t \in T$
$wrSet_t : L \pfun V$, initially empty for all $t \in T$

Transition relation:

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

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

$inv_t(\texttt{TMRead}(l))$
Pre: $pc_t =$ ready
Eff: $pc_t :=$ doRead(l)

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

$inv_t(\texttt{TMWrite}(l, v))$
Pre: $pc_t =$ ready
Eff: $pc_t :=$ doWrite(l, v)

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

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

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

$inv_t(\texttt{cancel})$
Pre: $pc_t =$ ready
Eff: $pc_t :=$ cancelPending

$resp_t(\texttt{abort})$
Pre: $pc_t \notin \{$notStarted, ready, commitResp, committed, aborted$\}$
Eff: $pc_t :=$ aborted

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

$\texttt{DoCommitWriter}_t$
Pre: $pc_t =$ doCommit
$readCons(latestMem, rdSet_t)$
Eff: $pc_t :=$ commitResp
$memories := memories \cup \{maxIdx + 1 \mapsto (latestMem \oplus wrSet_t)\}$

$\texttt{DoRead}_t(l, n)$
Pre: $pc_t =$ doRead(l)
$l \in \mathrm{dom}(wrSet_t) \vee validIdx(t, n)$
Eff: if $l \in \mathrm{dom}(wrSet_t)$ then
$pc_t :=$ readResp$(wrSet_t(l))$
else
$v := memories(n)(l)$
$pc_t :=$ readResp(v)
$rdSet_t := rdSet_t \oplus \{l \to v\}$

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

where

$$
\begin{aligned}
PCExternal &\mathrel{\widehat{=}} \{\text{notStarted, ready, commitResp, writeResp, committed, aborted}\} \cup \{\text{readResp}(v) \bullet v \in V\} \\
PCVal &\mathrel{\widehat{=}} PCExternal \cup \{\text{beginPending, doCommit, cancelPending}\} \\
&\quad \cup \{\text{doRead}(l) \bullet l \in L\} \cup \{\text{doWrite}(l, v) \bullet l \in L, v \in V\} \\
maxIdx &\mathrel{\widehat{=}} max(\mathrm{dom}(memories)) \\
latestMem &\mathrel{\widehat{=}} memories(maxIdx) \\
readCons(mem, rdSet) &\mathrel{\widehat{=}} rdSet \subseteq mem \\
validIdx(t, n) &\mathrel{\widehat{=}} beginIdx_t \leq n \leq maxIdx \wedge readCons(memories(n), rdSet_t)
\end{aligned}
$$

Fig. 4. The state space and transition relation of TMS2

TMS2 ensures that all transactions satisfy two constraints.

1. The read set of every transaction is consistent with at least one version of memory in $memories$ (the *consistency* constraint). This ensures that there is at least one valid serialization of all committing transactions.
2. The version of memory against which consistency is judged is the latest memory at some point during the transaction's execution (the real-time constraint). This ensures consistency with respect to the real-time order of the transactions.

State variables:
$glb : \mathbb{N}$, initially 0
$mem : L \to V$, initially satisfying $initMem(mem)$
$writer : T \cup \{\bot\}$, auxiliary variable, initially $writer = \bot$
$pc_t : PCExternal \cup Lines$, for each $t \in T$, initially $pc_t =$ notStarted, for all $t \in T$
$loc_t : \mathbb{N}$, for each $t \in T$, unconstrained initially
$addr_t : L$, for each $t \in T$, unconstrained initially
$val_t : V$, for each $t \in T$, unconstrained initially

Example transitions:

inv_t(TMBegin)
Pre: $pc_t =$ notStarted
Eff: $pc_t :=$ begin1

$resp_t$(TMBegin)
Pre: $pc_t =$ beginResp
Eff: $pc_t :=$ ready

write2$_t$
Pre: $pc_t =$ write2
Eff: if $loc_t = glb$ then
  $pc_t :=$ write4
  $glb := loc_t + 1$
  $writer := t$
else
  $pc_t :=$ Aborting

commit2$_t$
Pre: $pc_t =$ commit2
Eff: $pc_t :=$ commitResp
  $glb := loc_t + 1$
  $writer := \bot$

Fig. 5. The state space and selected transitions of the TML automaton

We assume $maxIdx$ denotes the largest $memories$ index, and $latestMem$ denotes the most recent version of memory. For a memory state $mem : L \to V$ and read set $rdSet : L \to V$, we assume $readCons(mem, rdSet)$ holds whenever $rdSet$ is consistent with mem. Finally, we assume predicate $validIdx(t, n)$ holds iff n is a *valid* index for a $t \in T$, i.e., n is between the maximum index of $memories$ when t began (recorded in the $beginIdx_t$ variable) and the current maximum index of $memories$, and furthermore, the contents of t's read set is consistent with $memories(n)$. This ensures that every transaction satisfies both the consistency and real-time constraints at every point before the transaction commits. Note that if the location l being read is in the transaction's write set, then TMS2 returns the value that was written.

With the exception of inv_t(TMBegin), external actions only depend on and modify the *program counter* variable pc_t. Collectively, they ensure that all transactions are well-formed, and that each "do" action occurs between the appropriate invocation and response. Additionally, action inv_t(TMBegin) sets $beginIdx_t$ to the maximum index in $memories$; this will be the earliest version of memory that can be read by non self-referencing reads of transaction t. The $resp_t$(abort) action is enabled whenever t has a pending operation (that is, whenever t has made an invocation without a matching response), but has not committed. Note that after the inv_t(cancel) action, no other action is possible except $resp_t$(abort).

The DoWrite$_t(l, v)$ action simply adds a mapping $l \mapsto v$ as a pending write in the write set of t. The precondition of action DoRead$_t(l, n)$ ensures that either the location l is self-referencing (i.e., l is in the write set of t), in which case it returns the value for l in $wrSet_t$, or n is a valid index for t (i.e., $validIdx(t, n)$ holds), in which case it returns the value at l in $memories(n)$. Read-only transactions may end by executing the DoCommitReadOnly$_t(n)$ action, while committing writer transactions take the DoCommitWriter$_t$ action.

The real-time ordering constraint for TMS2 is implicitly imposed by the preconditions of the DoCommitReadOnly$_t(n)$ and DoCommitWriter$_t$ actions. Action DoCommitReadOnly$_t(n)$ requires that n is a valid index for transaction t, whereas DoCommitWriter$_t$ requires that t's read set is consistent with the last element of $memories$.

4.3. The TML algorithm as an IOA

In this section, we describe how we model TML as an IOA. Fig. 5 presents the states of TML, and some example transitions. TML has the same external actions as TMS2, excluding the inv_t(cancel) actions (TML does not support cancellation). Each atomic statement of TML is modelled using an internal action.

We model all TML's local variables using functions over T. TML also has a shared glb variable of type $\mathbb{N}$, and a variable mem, which is a function from locations L to values V, representing the memory state.

TML uses the glb variable to implement a locking protocol. To assist reasoning about this protocol, we introduce an auxiliary variable $writer$ to record which transaction owns the lock (this variable takes the special value $\perp$ when no transaction owns the lock.) When transaction t successfully executes the CAS at line `W2`, $writer$ is set to t. When transaction t executes the write at line `C2`, $writer$ is set to $\perp$.

As with $TMS2$, the TML automaton uses a program-counter variable pc_t for each $t \in T$, to ensure that each transaction is well-formed, and that internal actions occur within the appropriate operation. In TML, the external actions are enabled by the same program-counter values as in TMS2 (values from the set $PCExternal$). There is also one program-counter value for each line in the pseudocode presented in Fig. 1.

4.4. The simulation proof

In this section, we describe the simulation relation used in the Isabelle proof. All Isabelle theory files related to this proof may be downloaded from the URL [TML16].

Definition 4.2 requires that for each internal transition of TML, either the concrete post-state is related to the abstract pre-state, or else there is some internal transition of TMS2 such that the concrete post-state is related to the abstract post-state. In the first case, we say that the concrete transition is a *stutter step*. In the second case, we say that the concrete transition *simulates* the abstract step. Our simulation relation must enable us to prove that when a concrete transition simulates an abstract transition, the precondition of the abstract transition is satisfied. We first motivate our simulation relation in terms of this requirement. We then describe in detail how concrete steps are matched with abstract steps and how we prove that the preconditions of the simulated steps are satisfied.

Our simulation relation is divided into two relatons: a *global relation* $globalRel$, and a transactional relation $txnRel$. The global relation describes how the shared state of the two automata are related, and the transaction relation specifies the relationship between the state of each transaction in the concrete automaton, and that of the transaction in the abstract automaton. The simulation relation itself is then

$$simRel(cs, as) = globalRel(cs, as) \wedge \forall\, t \in T \bullet txnRel(cs, as, t)$$

We first describe $globalRel$. Recall that TMS2 features a sequence of memory states, one for each writing transaction that has completed so far. The memory state specified by the sequence's $maxIdx$ is the latest memory. The memory of TML is this latest memory, except that if there is a writing transaction, then that transaction's writes have already been applied. This is because in TML, each write is applied immediately to the shared memory, whereas TMS2 defers the write. Thus, $globalRel$ specifies that for related concrete and abstract states cs and as, we have

$$cs.mem = as.latestMem \oplus writes(cs, as) \tag{1}$$

where

$$writes(cs, as) \mathrel{\widehat{=}} \begin{cases} \emptyset & \text{if } cs.writer = \perp \\ as.wrSet_{cs.writer} & \text{otherwise} \end{cases}$$

In the simulation proof, the role of (1) is to ensure that each `read1` action reads the appropriate value from the shared memory.

Recall that the preconditions of the $\texttt{DoRead}_t(l, n)$ and $\texttt{DoCommitReadOnly}_t(n)$ actions require that n be a valid index. We call these actions *validation actions*. Our simulation relation must be strong enough to allow us to verify that these conditions hold whenever TML takes a step that simulates a validation action. To achieve this, we exploit an association between the values taken by glb and those taken by $maxIdx$ in TMS2. Note that glb is incremented twice for each successful write transaction, and therefore the number of successful write transactions is $floor(glb/2)$. We define the *write-count* function, which for each $n \in \mathbb{N}$ yields the number of successful write transactions in an execution of TML if $cs.glb = n$ in the last state:

$$writeCnt(n) \mathrel{\widehat{=}} floor(n/2)$$

As we shall see, $writeCnt$ is used to relate $cs.glb$ with $as.maxIdx$, as well as each $cs.loc_t$ with $as.beginIdx_t$. Recall that in/ TMS2, $maxIdx$ is incremented once for each successful write transaction. Thus, $globalRel$ specifies that

$$writeCnt(cs.glb) = maxIdx(as) \tag{2}$$

$globalRel$ is just the conjunction of 1 and 2.

We turn now to $txnRel$. Let an *in-flight transaction* be any transaction t that has executed the $resp_t(\texttt{TMBegin})$ action but not executed either of the $resp_t(\texttt{TMEnd})$ or $resp_t(\texttt{abort})$ actions. $txnRel$ specifies that each in-flight transaction t satisfies the following properties:

$$as.beginIdx_t \leq writeCnt(cs.loc_t) \tag{3}$$

$$readCons(as.memories(writeCnt(cs.loc_t)), as.readSet_t) \tag{4}$$

These properties enable us to prove $as.validIdx(t, cs.loc_t)$ for all in-flight transactions t. To see this, first observe that in TML, every in-flight transaction satisfies $cs.loc_t \leq cs.glb$, and as described above, $writeCnt(cs.glb) = as.maxIdx$. Therefore,

$$writeCnt(cs.loc_t) \leq as.maxIdx \tag{5}$$

Together, (3), (4) and (5) imply $as.validIdx(t, cs.loc_t)$ for all in-flight transactions t.

Recall that the precondition of the $\texttt{DoCommitReadOnly}_t(n)$ action requires that t's write set be empty, and the precondition of the $\texttt{DoCommitWriter}_t$ action requires that the write set be nonempty. To handle this we exploit the fact that in TML, writing transactions have an odd loc value (at least after the first write has completed). $txnRel$ requires that each in-flight transaction satisfies the following equivalence, except during the interval between when a write transaction successfully executes the compare-and-swap at line `W2` and executes the subsequent write at line `W5`:

$$even(cs.loc_t) \iff as.wrSet_t = \emptyset \tag{6}$$

Because TML uses the parity of loc_t to determine whether a transaction is read-only, (6) enables us to prove the appropriate precondition when TML simulates a commit action.

In TML, when an in-flight transaction t has an odd loc_t value, it is the unique writing transaction, and $loc_t = glb$:

$$odd(cs.loc_t) \Rightarrow cs.writer = t \wedge cs.loc_t = cs.glb \tag{7}$$

As we describe below, this invariant enables us to prove that each writing transaction t satisfies the precondition of $\texttt{DoCommitWriter}_t$ when it commits, and that Property (1) is preserved during the actions $\texttt{write5}_t$ (when the transaction writes to the shared memory) and `commit2` (when the (writing) transaction commits).

All of TMS2's preconditions involve an assertion about the value of pc_t. Therefore, our simulation relation needs to constrain the relationship between $cs.pc_t$ and $as.pc_t$. This involves specifying an abstract program counter value for each possible concrete program counter value. For example, $txnRel(cs, as, t)$ implies

$$cs.pc_t = \text{notStarted} \Rightarrow as.pc_t = \text{notStarted} \tag{8}$$

$$cs.pc_t = \texttt{read1} \Rightarrow as.pc_t = \text{doRead}(cs.addr_t) \tag{9}$$

$$cs.pc_t = \texttt{write5} \Rightarrow as.pc_t = \text{doWrite}(cs.addr_t, cs.val_t) \tag{10}$$

$$cs.pc_t \in \{\texttt{commit1}, \texttt{commit2}\} \Rightarrow as.pc_t = \text{doCommit} \tag{11}$$

We are now ready to describe how concrete transitions are matched with abstract transitions. We do so using a *step-correspondence* function, denoted sc, that takes a concrete state, a transaction and an internal action, and returns an appropriate abstract internal action, or else $\perp$, indicating that the step of TML is a stutter step. In the list below, we describe the cases when $sc(cs, t, a) \neq \perp$. For each case, we explain how to prove that the precondition of the abstract action holds.

- A read operation takes effect when a transaction executes `R1` when `glb` has not been modified since the transaction began. Thus, if $a = \texttt{read1}$ and $cs.loc_t = cs.glb$, then $sc(cs, t, a) = \texttt{DoRead}_t(cs.addr_t, writeCnt(cs.loc_t))$. At this point t is in-flight transaction. Therefore, as described above, $txnRel$ guarantees that $validIdx(t, writeCnt(cs.loc_t))$. Property (9) ensures that $as.pc_t$ has the correct value.
- A write operation takes effect when when a transaction executes the write at line `W5`. Thus, if $a = \texttt{write5}$ then $sc(cs, t, a) = \texttt{DoWrite}_t(cs.addr_t, cs.val_t)$. Property (10) ensures that $as.pc_t$ has the correct value.
- When a transaction begins the commit operation with an even `loc` variable, then it is a read-only transaction. Therefore, when $a = \texttt{commit1}$ and $cs.loc_t$ is even, $sc(cs, t, a) = \texttt{DoCommitReadOnly}_t(writeCnt(cs.loc_t))$. Again, $txnRel$ guarantees that $validIdx(t, writeCnt(cs.loc_t))$. Property (6) ensures that t's write set is empty. Property (11) ensures that $as.pc_t$ has the correct value.
- If a transaction begins the commit operation with an odd `loc` variable, it is a writing transaction that owns the TML lock. After executing the write at line `C2` that releases the TML lock, the transaction's writes become visible to the other transactions, so the transaction takes effect at this point. Thus, when $a = \texttt{commit2}$,

$sc(cs, t, a) = \texttt{DoCommitWriter}_t$. The precondition of $\texttt{DoCommitWriter}_t$ requires that $readCons(as.latestMem, as.readSet_t)$. We already know that $validIdx(t, writeCnt(cs.loc_t))$ and that $writeCnt(cs.glb) = as.maxIdx$. By Invariant (7) of TML is that when the writing transaction t commits, $cs.glb = cs.loc_t$. Therefore, $validIdx(t, as.maxIdx)$ and thus $readCons(as.latestMem, as.rdSet_t)$. Property (6) ensures that t's write set is nonempty. Finally, Property (11) ensures that $as.pc_t$ has the correct value as usual.

In all other cases $sc(cs, t, a) = \bot$.

We have not yet discussed how we show that the simulation relation is preserved across all steps of TML. These proofs are largely routine. To give a flavour of how these proofs work, we prove that Property (1) is preserved across all transitions. We consider an arbitrary transition $(cs, a, cs') \in trans(TML)$ and abstract state as where $simRel(cs, as)$. The variables that appear in Property (1) are $cs.mem$, $cs.writer$, $as.memories$, and $as.wrSet_{cs.writer}$. None of these variables can be changed except when $a = \texttt{commit2}_t$, $a = \texttt{write2}_t$ when $cs.loc_t = cs.glb$ or $a = \texttt{write5}_t$ for some $t \in T$. We consider each case in turn.

- When $a = \texttt{commit2}_t$, $cs'.writer = \bot$ and $sc(cs, a, t) = \texttt{DoCommitWriter}_t$. Therefore, $writes(cs') = \emptyset$, so we must prove that $cs'.mem = as'.latestMem$, where as' is the post-state of the abstract transition. But $cs.writer = t$ by Property (7) and the fact that $cs.loc_t$ is odd at this point in the code. Thus,

$$\begin{aligned}
&cs'.mem \\
&= cs.mem && \text{by the transition relation of TML} \\
&= as.latestMem \oplus as.wrSet_{cs.writer} && \text{Property (1)} \\
&= as.latestMem \oplus as.wrSet_t && \text{since } t = cs.writer \\
&= as'.latestMem && \text{by the transition relation of TMS2}
\end{aligned}$$

  as required.
- TML has the invariant that $cs.writer = \bot \iff even(cs.glb)$. Therefore, when $a = \texttt{write2}_t$ and $cs.loc_t = cs.glb$, $cs.loc_t$ is even and therefore $cs.writer = \bot$ and thus $writes(cs) = \emptyset$. Furthermore, $as.wrSet_t = \emptyset$ by Property (6). Thus

$$\begin{aligned}
&cs'.mem \\
&= cs.mem && \text{by the transition relation of TML} \\
&= as.latestMem \oplus writes(cs) && \text{by Property (1)} \\
&= as.latestMem && \text{since } writes(cs) = \emptyset \\
&= as.latestMem \oplus as.wrSet_t && \text{since } as.wrSet_t = \emptyset \\
&= as.latestMem \oplus writes(cs') && \text{since } cs'.writer = t
\end{aligned}$$

  as required.
- When $a = \texttt{write5}_t$, $cs.loc_t$ is odd and hence $cs.writer = t$ by Property (7). $sc(cs, a, t) = \texttt{DoWrite}_t(cs.addr_t, cs.val_t)$, so $as'.wrSet = as.wrSet \oplus \{cs.addr_t \mapsto cs.val_t\}$. Thus

$$\begin{aligned}
&cs'.mem \\
&= cs.mem \oplus \{cs.addr_t \mapsto cs.val_t\} && \text{by the transition relation of TML} \\
&= (as.latestMem \oplus as.wrSet_t) \oplus \{cs.addr_t \mapsto cs.val_t\} && \text{by Property (1) and } cs.writer = t \\
&= as.latestMem \oplus (as.wrSet_t \oplus \{cs.addr_t \mapsto cs.val_t\}) \\
&= as.latestMem \oplus as'.wrSet_t && as'.wrSet = as.wrSet \oplus \{cs.addr_t \mapsto cs.val_t\} \\
&= as.latestMem \oplus writes(cs') && \text{since } cs.writer = cs'.writer = t \\
&= as'.latestMem \oplus writes(cs') && \text{by transition relation of TMS2}
\end{aligned}$$

  as required.

Mechanization We have mechanized the TMS2 automaton, and the TML model in Isabelle. As has been mentioned, we used the IO automaton model of Müller [Mül98]. We have also proved in Isabelle that that $simRel$ is a forward simulation, and that the TML automaton satisfies all the necessary invariants. We have not mechanized a proof that every TMS2 trace is opaque, because this proof has already been mechanized in [LLM12b].

Formalising TML in Isabelle and completing these proofs took one person 8 working days, working on the proof about 70% of full time. The formalisation used a pre-existing model of the TMS2 automaton in Isabelle. The proofs were developed using the *Isar* proof language [Wen02]. Isar provides facilities for decomposing a proof into cases, which is important for quickly identifying the aspects of a proof that require human intervention. Once this stucture was established, it was straightforward to determine which simplification rules and lemmas should be applied in each case. Typically, this amounted to choosing an appropriate set of definitions to expand. The simulation and invariant proofs themselves only required more sophisticated lemmas on 10 occassions. Writing these proofs was routine, though somewhat time consuming.

5. Comparison of the techniques

In this section, we compare the two verifications of TML. Recall that the linearizability method (i.e., Method 1) proves inductively that for each history of TML there is an opaque linearized history, and thus that the TML history is opaque. On the other hand, the simulation method (i.e., Method 2) proves the existence of a forward simulation from an IO automaton representing TML to the TMS2 automaton, whose traces are opaque.

Specifications The abstract specifications used in the two methods differ in one important respect. In TMS2 (and in conventional definitions of opacity), actions are indexed by transaction identifiers. Threads or processes are not an explicit part of the model. In the definition of opacity given in this paper, however, events are indexed by processes, and transactions are modelled as sequences of transactional operations satisfying appropriate well-formedness conditions (see Sect. 2.2). The key difference is that one process can execute several transactions, but each transaction identifier can only be associated with one transaction.

The linearizability method verifies TML with respect to the definition of opacity of Sect. 2.3. This choice is not fundamental to the linearizability proof. A linearizability-based method could be completed using the same techniques, but where events are indexed by transaction identifiers (or equivalently, where each process executes at most one transaction). Likewise, TMS2 could be adpated to support process-indexed events: transaction-indexed variables would become process indexed variables, and these variables would be reset when each transaction ended.

The fact that processes can execute several transactions does introduce complexity to the proof. We need mechanisms to resolve runs (and histories) into transactions, and to paste transactions back together into runs. This complexity can be observed in the definition of *transaction* in Sect. 2.2 and the need for the function $tseq$.

Because of its relative simplicity, the transaction-indexed model used in the simulation method is preferable when considering STM systems in isolation. However, the process-indexed model is more faithful to an STM interface that one would expect to find in a multithreaded library or programming language, and is uniform with models used to represent linearizable concurrent objects. For this reason, the process-indexed model may be preferable when considering the behaviour of clients of a transactional memory, especially when the transactional memory is composed with other concurrency abstractions, such as linearizable objects.

Proof decomposition Both methods employ an intermediate layer between the behaviour of TML and opacity. However, this layer is different in each method. In the simulation method, this intermediate layer is TMS2, an automaton whose histories are already known to be opaque (see Fig. 3). In order to show opacity for TML, it was only necessary to prove the existence of a simulation from TML to TMS2. The simulation proof benefits by decomposing the proof of opacity into two independent proofs.

In the linearizability proof method, the intermediate layer is the set of alternating histories that linearize histories generated by TML (see Sect. 3.3). We show that these alternating histories are opaque (Sect. 3.4). However, this part of the proof depends intrinsically on invariants of TML itself, and thus there is no real decomposition. It is important to note that the linearized histories do not constitute an intermediate *specification* for transactional memory, in the same way as TMS2 is a self-contained specification.

Generality of the method In the linearizability method, each run event is given a semantics in terms of its effect on memory states, based on the eager-writes of TML (see Sect. 3.4). Unfortunately, this simple semantics cannot be used to model the behaviour of an STM implementation that uses deferred updates, so the applicability of this semantics is limited to STM implementations with eager updates. In order to support deferred updates, a new and more complicated semantics for run events would need to be developed. This semantics would likely involve recording write sets for each transaction, and applying these write sets to a shared memory during commits, in a manner similar to TMS2's `DoWrite` action. This extra machinery would complicate the proof that the existence of a valid run guarantees opacity of the corresponding alternating history (that is, it would be more difficult to prove the analogue of Theorem 3.2).

In contrast, TMS2 can be used to verify algorithms with either deferred- or eager-update semantics. As has been noted, there is already a published simulation proof that the *NORecs* algorithm (which employs deferred updates) implements TMS2 [LLM12a], and we believe that several other STM algorithms [DSS06, SMvP08, HLMSI03] could be shown to implement TMS2 using simulation methods. For these reasons, we believe that TMS2 and simulation can be used to verify a large and important class algorithms.

We should note that forward simulation alone is not always enough to prove trace inclusion. In some cases, the forward simulation technique that we have used here would need to be replaced or extended by *backward simulation* [LV95]. While being closely related, backward simulations have a different flavour, and are sometimes counter-intuitive. However, forward and backward simulations together provide a complete proof method for trace inclusion. That is, if all traces of automaton C are also traces of automaton A, then there is some intermediate automaton I such that there is a forward simulation from C to I, and a backward simulation from I to A [LV95].

There is another obstacle to straightforwardly generalising the simulation method. TMS2 is strictly stronger than opacity (see [DGLM13] for a discussion). Although at the time of writing we are not aware of any, there may be implementations of transactional memory that are opaque, but do not implement TMS2. It is possible to use the simulation methodology in such cases, although it may be more difficult. Lesani et al. [LLM12b] present an *opacity automaton* whose traces are precisely the opaque histories. Because of the completeness of forward and backward simulation, proving the opacity of any algorithm can be reduced to proving the existence of appropriate simulations between an IOA model of the algorithm and the opacity automaton.

Inductive structure Both verification methods share a common structure, a structure that is characteristic of refinement proofs. Both work by constructing a certain abstract object (i.e., specification), inductively along each execution of TML. In the simulation proof, this object is a state of TMS2 satisfying the simulation relation. In the linearizability proof, the abstract object is the run witnessing the opacity of the TML history. In each case, the existence of the abstract object is sufficient to guarantee the correctness of the execution so far. In this sense, both proofs work the same way. However, the two abstract objects are very different. We now consider which of these abstract objects is simpler to reason about.

Both proofs must address the problem of demonstrating that opacity's real-time order constraint is satisfied. To accomplish this, the simulation proof depends heavily on the correctness of TMS2. Recall from Sect. 4.2 that TMS2 guarantees that each nonaborting transaction is consistent with respect to a memory state that was the latest state sometime during the execution of the transaction, and this is achieved using the $beginIdx$ variable. Property (3) ensures that $writeCnt(cs.loc_t)$ is at least as great as $beginIdx_t$, and therefore may be a valid index for t. Property (3) is simple to state and it is easy to prove that it is preserved. The corresponding properties in the linearizability proof are significantly more complicated. The key invariants for proving the real-time constraint in the linearizability proof are INV3 and INV4. Both invariants involve an assertion about the order in which transactions occur in the run rs, and proving their preservation involves reasoning about sequences rather than natural numbers. Furthermore, preservation of INV3 necessitates reordering of transactions in the run sequence, during commit and abort steps.

Both proofs must address the problem of demonstrating that TML's shared memory contains appropriate values at each location (i.e., that the value at each location can be legally returned by any read operation of any transaction such that $loc = glb$). The simulation proof achieves this using Property (1), and the linearizability proof does so using INV5. Ideally, Property (1) would be a simple equality between TML's memory and the $latestMem$ of TMS2. The fact that TML uses eager updates, but TMS2 uses deferred updates necessitates the more complicated formulation. INV5 is simpler, in that the last state of the validating sequence σ is simply the shared memory of TML.

Both proofs must address a peculiarity of TML's behaviour: a read-only transaction whose loc variable is strictly less than glb may still commit successfully (so long as it does not attempt a read or write operation). This is possible even when multiple conflicting writing transactions may have complete since the read-only transaction began. This phenomenon is handled straightforwardly in the simulation proof. In TMS2 read-only transactions may be validated against an old version of the memory when they commit. As discussed in Sect. 3.4, this is a delicate case in the linearizability proof, requiring special care in constructing the new sequence of memory states.

INV2 and Property (7) both express the same idea: there is at most one transaction with an odd loc variable (the writing transaction), and the value of this variable is equal to glb. However, Property (7) does this using the $writer$ auxiliary variable. Whereas INV2 expresses this using a more complicated formulation involving ordering within the run sequence: the writing transaction is the last transaction in rs.

Mechanisation Neither proof is tied to a specific proof assistant, both are based on formal specifications using higher-order logic. The linearizability proof was done with KIV, but could easily be ported to Isabelle. The second

proof was done independently with Isabelle and KIV, to have a comparison between the different approaches to specification and the different proof styles used in the tools. Sect. 4 describes the Isabelle proof.

The total time to complete the linearizability proof with KIV was about 4 weeks, of which about 2 weeks were spent modelling and verifying the TML algorithm itself. The simulation proof required around a week for both Isabelle and KIV.

As the KIV simulation proof was developed after the linearizability proof, the reduction in effort is partly attributable to the familiarity with TML and its invariants gained during the linearizability proof. In the case of the KIV simulation proof some part of the speedup is also due to reusing parts of the specification of TML. However, even taking this into account we believe the simulation proof required less that 40% of the effort required for the linearizability proof proper. The main part of the gain can be attributed to the fact that the TMS2 automaton does not require certain auxiliary definitions used to define opacity, that employ quantifiers. For example, transactions require the *existence* of processes, which have executed specific events in the history; the semantics requires the *existence* of a suitable memory sequence; preserving the real time constraint requires that *all* pairs of transactions are correctly ordered. Also, all of the invariants INV1-INV6 involve quantification over all transactions.

To deal with such quantified properties in the linearizability proof, it was necessary to define various rewrite rules for use in the main proof. Without this, unfolding the quantified definitions in the main proof leads to large and unreadable formulae, as well as the need to instantiate quantifiers manually. Defining rewrite rules for quantified definitions was a main part of the proof effort of the linearizability approach. Altogether the proof in KIV required 80 lemmas, 150 rewrite rules (that were not already in the library of predefined data types).

In contrast, the main proof of the step correspondence condition of the simulation approach involves almost no quantified definitions, except for the three top-level quantifiers: the simulation relation for *each* transaction must be preserved by *every* step of the algorithm, and a suitable abstract state as' must *exist* to make the diagram commute. Both the proof in Isabelle and KIV therefore first define lemmas that remove of these top-level quantifiers.

For the KIV proof of the simulation approach this resulted in four lemmas, and was already sufficient: the proofs of the four lemmas just blindly unfold definitions and require just 5 rewrite rules. Altogether 248 proof steps with 91 interactions were needed.

The Isabelle and KIV simulation proofs follow the same ideas, although there are many technical differences. Here, we only mention the most important two. Full details can be found on the Web page [TML16].

- The KIV proof uses the linearization points R2 and W5 (c.f. Fig. 1) from the linearizability proof as the steps that simulate DoRead/DoWrite, while the Isabelle proof uses R1 for DoRead.
- The KIV proof avoids the need to compute the parameter n of DoRead as $floor(glb/2)$ by existentially quantifying over n in the TMS2 automaton. The simulation also just asserts the existence of a valid n. The price to pay is that DoRead is the only non-deterministic transition, and its simulation proof needs an extra lemma.

In summary the Isabelle proof has more structure, (e.g. it uses an intermediate layer of deterministic automata, and proves invariants separately from the simulation), at the price of defining more lemmas.

Conclusions We believe that the simulation proof has several advantages over the linearizability proof. We have argued that it is simpler and required less time to complete. This simplicity derives from the fact that the proof used an existing intermediate specification, known to be opaque. The simulation proof avoids reasoning directly about histories, and avoids explicitly maintaining a serialisation of transactions.

As we have noted, the question of whether events should be indexed by process or transaction identifiers is independent of the proof method. Some of the complexity in the linearizability proof derives from the choice to index events with processes, each of which can execute several transactions. However, this model may be preferable when considering clients of transactional memory, or when transactional memory is composed with other concurrent objects.

Based on this particular study, we offer no suggestions about the choice of proof assistant. Clearly the best choice of proof assistant depends on the experience that the human prover already has, and on the level of help and advice that the prover can obtain from colleagues and collaborators.

6. Conclusions

There are many notions of correctness for STMs [Les14, GK10]. Of these, opacity is an easy-to-understand notion that ensures all reads are consistent with committed writing transactions. We have developed a proof

method for, and verified opacity of, a transactional mutex lock implementation. Many definitions of opacity in the literature require an explicit mention of the permutations on histories, which would make proofs significantly more complex. Our formalization has avoided the explicit use of permutations.

Opacity defines correctness in terms of histories generated by interleaving STM operations as well as statements within the operations. Our method simplifies proof of opacity by reformulating opacity in terms of runs, and proving opacity of the runs. A run allows interleaving of operations, but each operation is treated as being atomic, and hence, the statements within an operation are not interleaved. Linearizability is used to justify replacing an interleaved history by an alternating one (Theorem 3.1), while Theorem 3.2 justifies proving opacity of an alternating history by proving opacity of the run corresponding to the history.

Our second proof method is based a definition of opacity specified in terms of IOA. This is particularly interesting for verification, as it readily gives us a formal specification which can be used as an abstract level in a refinement-based proof. Refinement is also the way of comparing definitions in this case: TMS2 is known to refine TMS1 and (an IOA version of) opacity [LLM12b].

Although there are several works comparing and contrasting different correctness conditions for STM (including opacity) (e.g., [DGLM13, LLM12b, AGHR14]), there only a handful of papers that consider verification of the STM implementations themselves. A model checking approach is presented in [GHS10], however, the technique only considers *conflicts* between read and write operations in different transactions. More recently, Lesani has considered opacity verification of numerous algorithms [Les14], which includes techniques for reducing the problem of proving opacity into one of verifying a number of simpler invariants on the orders of events [LP14]. However, these decomposed invariants apply directly to the interleaved histories of the implementation at hand, as opposed to our method that performs a decomposition via runs.

In ongoing work, we are developing in Isabelle a framework for mechanically verifying TM algorithms. As discussed in Sect. 1, Lesani et al. [LLM12a] have pursued similar goals; namely, a TM verification framework for the PVS proof assistant based on IOA and simulation, and the verification of the NOrec STM algorithm. There are several technical differences between the two approaches. Our Isabelle development uses the existing Isabelle IOA formalization, which supports reasoning about both safety and liveness properties. Lesani et al. [LLM12a] have developed their own IOA formalization that only supports safety properties. They employ a more hierarchical proof structure, introducing three intermediate automata between TMS2 and a low level NOrec model. We prove a direct simulation from TML to TMS2. These extra layers are at least partly motivated by the complexity of the NOrec algorithm, compared with TML. However, there is considerable overhead in defining intermediate automata. Understanding how best to hierarchically decompose proofs, while reducing overall proof effort, is part of our ongoing work.

Finally, [LLM12a] handles actions and traces differently to us, and to what is standard for IOA. In a typed setting, it seems natural that each automaton has its own type of actions. However, actions in different types cannot be equal. Therefore, if each automaton has its own type, the traces of one automaton can never be traces of another. Lesani et al. [LLM12a] resolve this issue by equipping each automaton with a *view* function that maps the automaton's actions into a set of *events*, which are shared between automata. In our solution, we define labelled transition systems whose actions are from a disjoint union of internal and external actions. We transform this LTS into an IOA whose actions are from a disjoint union of the external actions and a type with a unique *tau* element. In the IOA transition relation, these tau transitions are just the transitions of the underlying LTS where the internal action is hidden by existential quantification. Understanding the relative merits of these two approaches requires further study. However, our aproach fits directly into the existing IOA framework.

References

[AGHR13] Attiya H, Gotsman A, Hans S, Rinetzky N (2013) A programming language perspective on transactional memory consistency. In: Fatourou P, Taubenfeld G (eds) PODC'13. ACM, pp 309–318

[AGHR14] Attiya H, Gotsman A, Hans S, Rinetzky N (2014) Safety of live transactions in transactional memory: TMS is necessary and sufficient. In: Kuhn F (ed) DISC, volume 8784 of LNCS. Springer, pp 376–390

[ASP16] Anand AS, Shyamasundar RK, Peri S (2016) Opacity proof for CaPR+ algorithm. In: Proceedings of the 17th international conference on distributed computing and networking, ICDCN '16, New York, NY, USA. ACM, pp 16:1–16:4

[COC+15] Cristal A, Kulahcioglu Ozkan B, Cohen E, Kestor G, Kuru I, Unsal OS, Tasiran S, Mutluergil SO, Elmas T (2015) Verification tools for transactional programs. In: Guerraoui R, Romano P (eds) Transactional memory. Foundations, algorithms, tools, and applications—COST Action Euro-TM IC1001, volume 8913 of lecture notes in computer science. Springer, pp 283–306

[COP+07] Cohen A, O'Leary JW, Pnueli A, Tuttle MR, Zuck LD (2007) Verifying correctness of transactional memories. In: FMCAD, Washington, DC, USA. IEEE Computer Society, pp 37–44

[DD15] Dongol B, Derrick J (2015) Verifying linearisability: a comparative survey. ACM Comput Surv 48(2):19

[DDS+10] Dalessandro L, Dice D, Scott ML, Shavit N, Spear MF (2010) Transactional mutex locks. In: D'Ambra P, Guarracino MR, Talia D (eds) Euro-Par (2), volume 6272 of LNCS. Springer, pp 2–13

[DDS+15] Derrick J, Dongol B, Schellhorn G, Travkin O, Wehrheim H (2015) Verifying opacity of a transactional mutex lock. In: FM, volume 9109 of LNCS. Springer, pp 161–177

[DGLM04] Doherty S, Groves L, Luchangco V, Moir M (2004) Formal verification of a practical lock-free queue algorithm. In: FORTE, volume 3235 of LNCS. Springer, pp 97–114

[DGLM13] Doherty S, Groves L, Luchangco V, Moir M (2013) Towards formally specifying and verifying transactional memory. Formal Asp Comput 25(5):769–799

[DSS06] Dice D, Shalev O, Shavit N (2006) Transactional locking II. In: Dolev S (ed) DISC, volume 4167 of LNCS. Springer, pp 194–208

[DSS10] Dalessandro L, Spear MF, Scott ML (2010) Norec: streamlining STM by abolishing ownership records. In: Govindarajan R, Padua DA, Hall MW (eds) PPoPP. ACM, pp 67–78

[DSW11] Derrick J, Schellhorn G, Wehrheim H (2011) Verifying linearisabilty with potential linearisation points. In: Proceedings formal methods (FM), LNCS 6664. Springer, pp 323–337

[EMM10] Emmi M, Majumdar R, Manevich R (2010) Parameterized verification of transactional memories. SIGPLAN Not 45(6):134–145

[EPS+14] Ernst G, Pfähler J, Schellhorn G, Haneberg D, Reif W (2015) KIV: overview and VerifyThis competition. Int J Softw Tools Technol Transfer 17(6):677–694. doi:10.1007/s10009-014-0308-3

[GHS08] Guerraoui R, Henzinger TA, Singh V (2008) Completeness and nondeterminism in model checking transactional memories. In: van Breugel F, Chechik M (eds) CONCUR. Springer, pp 21–35

[GHS10] Guerraoui R, Henzinger TA, Singh V (2010) Model checking transactional memories. Distrib Comput 22(3):129–145

[GK08] Guerraoui R, Kapalka M (2008) On the correctness of transactional memory. In: Chatterjee S, Scott ML (eds) PPOPP. ACM, pp 175–184

[GK10] Guerraoui R, Kapalka M (2010) Principles of transactional memory. Synthesis lectures on distributed computing theory. Morgan & Claypool Publishers, San Rafael

[HLMSI03] Herlihy M, Luchangco V, Moir M, Scherer III WN (2003) Software transactional memory for dynamic-sized data structures. In: PODC. ACM, pp 92–101

[HLR10] Harris T, Larus JR, Rajwar R (2010) Transactional memory. In: Synthesis lectures on computer architecture, 2nd edn. Morgan & Claypool Publishers, San Rafael

[HW90] Herlihy M, Wing JM (1990) Linearizability: a correctness condition for concurrent objects. ACM TOPLAS 12(3):463–492

[IR12] Imbs D, Raynal M (2012) Virtual world consistency: a condition for STM systems (with a versatile protocol with invisible read operations). Theor Comput Sci 444:113–127

[Lam79] Lamport L (1979) How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans Comput 28(9):690–691

[Les14] Lesani M (2014) On the correctness of transactional memory algorithms. Ph.D. thesis, UCLA

[LLM12a] Lesani M, Luchangco V, Moir M (2012) A framework for formally verifying software transactional memory algorithms. In: Koutny M, Ulidowski I (eds) CONCUR 2012. Springer, Berlin, pp 516–530

[LLM12b] Lesani M, Luchangco V, Moir M (2012) Putting opacity in its place. In: Workshop on the theory of transactional memory

[LP13] Lesani M, Palsberg J (2013) Proving non-opacity. In: Afek Y (ed) DISC, volume 8205 of LNCS. Springer, pp 106–120

[LP14] Lesani M, Palsberg J (2014) Decomposing opacity. In: Kuhn F (ed) DISC, volume 8784 of LNCS. Springer, pp 391–405

[LT87] Lynch NA, Tuttle MR (1987) Hierarchical correctness proofs for distributed algorithms. In: PODC, New York, NY, USA. ACM, pp 137–151

[LV95] Lynch N, Vaandrager F (1995) Forward and backward simulations. Inf Comput 121(2):214–233

[LZCF10] Li Y, Zhang Y, Chen Y-Y, Fu M (2010) Formal reasoning about lazy-STM programs. J Comput Sci Technol 25(4):841–852

[Mül98] Müller O (1998) I/O Automata and beyond: temporal logic and abstraction in Isabelle. In: Grundy J, Newey M (eds) TPHOLs. Springer, Berlin, pp 331–348

[NPW02] Nipkow T, Paulson LC, Wenzel M (2002) Isabelle/HOL— a proof assistant for higher-order logic, volume 2283 of LNCS. Springer

[ORS92] Owre S, Rushby JM, Shankar N (1992) PVS: A prototype verification system. In: Kapur D (ed) Automated deduction—CADE-11, 11th international conference on automated deduction, Saratoga Springs, NY, USA, June 15–18, 1992, proceedings, volume 607 of LNCS. Springer, pp 748–752

[Pap79] Papadimitriou CH (1979) The serializability of concurrent database updates. J ACM 26(4):631–653

[SDW14] Schellhorn G, Derrick J, Wehrheim H (2014) A sound and complete proof technique for linearizability of concurrent data structures. ACM Trans Comput Log 15(4):31:1–31:37

[SMvP08] Spear MF, Michael MM, von Praun C (2008) RingSTM: scalable transactions with a single atomic instruction. In: Proceedings of the twentieth annual symposium on parallelism in algorithms and architectures. ACM, pp 275–284

[TML16] Verification of opacity of a Transactional Mutex Lock with KIV and Isabelle, 2016. http://www.informatik.uni-augsburg.de/swt/projects/Opacity-TML.html
[Vaf07] Vafeiadis V (2007) Modular fine-grained concurrency verification. Ph.D. thesis, University of Cambridge
[Wen02] Wenzel M (2002) Isabelle/Isar-a versatile environment for human-readable formal proof documents. Ph.D. thesis, Institut für Informatik, Technische Universität München

Received 2 March 2016
Accepted in revised form 21 July 2017 by Frank de Boer, Nikolaj Bjorner, Andrew Butterfield and Jim Woodcock