Proving Opacity via Linearizability: A Sound and Complete Method

Alasdair Armstrong, Brijesh Dongol, Simon Doherty

Proving Opacity via Linearizability: A Sound and Complete Method

Alasdair Armstrong[1], Brijesh Dongol[1], and Simon Doherty[2]

[1] Brunel University London, UK
[2] University of Sheffield, UK

Abstract. Transactional memory (TM) is a mechanism that manages thread synchronisation on behalf of a programmer so that blocks of code execute with the illusion of atomicity. The main safety criterion for transactional memory is opacity, which defines conditions for serialising concurrent transactions.
Verifying opacity is complex because one must not only consider the orderings between fine-grained (and hence concurrent) transactional operations, but also between the transactions themselves. This paper presents a sound and complete method for proving opacity by decomposing the proof into two parts, so that each form of concurrency can be dealt with separately. Thus, in our method, verification involves a simple proof of opacity of a coarse-grained abstraction, and a proof of linearizability, a better-understood correctness condition. The most difficult part of these verifications is dealing with the fine-grained synchronization mechanisms of a given implementation; in our method these aspects are isolated to the linearizability proof. Our result makes it possible to leverage the many sophisticated techniques for proving linearizability that have been developed in recent years. We use our method to prove opacity of two algorithms from the literature. Furthermore, we show that our method extends naturally to weak memory models by showing that both these algorithms are opaque under the TSO memory model, which is the memory model of the (widely deployed) x86 family of processors. All our proofs have been mechanised, either in the Isabelle theorem prover or the PAT model checker.

1 Introduction

Transactional Memory (TM) 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 akin to database transactions [16]. Recent years have seen an explosion of interest in TM, leading to the implementation of TM libraries for many programming languages (including Java and C++), compiler support for TM (G++ 4.7) and hardware support (e.g., Intel's Haswell processor). This widespread adoption coupled with the complexity of TM implementations makes formal verification of TM an important problem.

The main safety condition for TM is *opacity* [14, 15], which defines conditions for serialising (concurrent) transactions into a sequential order and specifies which data values transactions may read. A direct proof of opacity must somehow construct an

appropriate serialisation of the transactions. This is complicated by the fact that transactions are not constrained to read the most recently committed value at any given address. Because of this, several "snapshots" of the transactional memory must be made available to each transaction.

This situation may be contrasted with the well-known correctness condition *linearizability* [17]. Unlike opacity, linearizability proofs only need to consider a single value of the abstract object. Operations never "look back in time" to some earlier state, and linearizability proofs are therefore less complex. Furthermore, there is a rich literature on the verification of linearizability (see [12] for a survey), whereas the verification of opacity has received much more limited attention. Techniques exist for verifying linearizability using data-refinement [10, 25], separation logic and other program logics [27, 6], and model-checking [23, 4, 5]. With the possible exception of data refinement, none of these techniques are available for the verification of opacity.

These observations motivate us to explore methods for recasting the problem of verifying opacity to that of verifying linearizability, and this paper presents one such method. Briefly, our method involves the construction of a *coarse-grained abstraction* (CGA) that serves as an intermediate specification between the TM implementation to be verified and opacity itself. Our method requires us to prove that this CGA is opaque. But, as we shall see, the CGA features a coarse grain of atomicity and a simplified state space, relative to the original implementation. These features make verifying opacity of the CGA very straightforward. Importantly, we do not need to consider the complex interleaving and fine-grained synchronisation mechanisms of the original implementation in this part of the proof. Our method also requires us to prove the linearizability of the original TM implementation where the CGA becomes the abstract specification. Only at this point is it necessary to consider the fine-grained synchronization of the actual TM implementation. But for this linearizability proof we can leverage the powerful techniques for verifying linearizability that have been developed in recent years.

We adapt a result from [9] to prove that our method is *sound*: any verification using our technique guarantees opacity of the original algorithm. We also show that our method is *complete*: for any opaque TM implementation, there must exist an opaque CGA, such that the original implementation is linearizable with respect to the CGA. We use our method to prove opacity of two TM implementations: the Transactional Mutex Lock [7], and the more sophisticated and practical NORec algorithm [8]. In addition, we show that our method extends to weak memory models: we verify opacity of both TML and NORec under TSO memory.

For full details of our mechanisations see our extended report [2], which includes all mechanisations, and further descriptions of our proofs.

2 Transactional Memory

In this section, we describe the interface provided by the TM abstraction, and give an example of a transactional memory algorithm: the simple but efficient Transactional Mutex Lock (TML) by Dalessandro *et al.* [7]. Then we formalise *opacity* as defined by Guerraoui and Kapalka [15]. Our formalisation mainly follows Attiya *et al.* [3],

Listing 1 The Transactional Mutex Lock (TML) algorithm

```
1:  procedure INIT
2:      glb ← 0
3:  procedure TXBegin_t
4:      do loc_t ← glb
5:      until even(loc_t)
6:  procedure TXCommit_t
7:      if odd(loc_t) then
8:          glb ← loc_t + 1
9:  procedure TXRead_t(a)
10:     v_t ← mem(a)
11:     if glb = loc_t then
12:         return v_t
13:     else abort
14: procedure TXWrite_t(a, v)
15:     if even(loc_t) then
16:         if !cas(&glb, loc_t, loc_t+1)
17:         then abort
18:         else loc_t++
19:     mem(a) ← v
```

but we explicitly include the prefix-closure constraint to ensure consistency with other accepted definitions [21, 15, 16].

To support transactional programming, TM provides a number of operations[3] to developers: operations to start (`TXBegin`) or to end a transaction (`TXCommit`), and operations to read or write shared data (`TXRead`, `TXWrite`). These operations can be invoked from within a program (possibly with some arguments, e.g., the address to be read) and then will return with a response. Except for operations that start transactions, all other operations can respond with a special *abort* value, thereby aborting the whole transaction.

Transactional Mutex Lock (TML). The TML algorithm is presented in Listing 1. It provides the four operations, but operation `TXCommit` in this algorithm never responds with abort. TML adopts a very strict policy for synchronisation among transactions: as soon as one transaction has successfully written to an address, other transactions running concurrently will be aborted if they subsequently invoke a `TXRead` or `TXWrite` operation on *any* address. For synchronisation, TML uses a global counter glb (initially 0), and each transaction t uses a local variable loc_t to store a local copy of glb. Variable glb records whether there is a *live writing transaction*, i.e., a transaction which has started, has neither committed nor aborted, and has executed a write operation. More precisely, glb is odd if there is a live writing transaction, and even otherwise. Initially, there are no live writing transactions and thus glb is even.

Histories. As is standard in the literature, opacity is defined over *histories*, which are sequences of *events* that record all interactions between the TM and its clients. Each event is either the invocation or response of some TM operation. Possible invocations and their matching response events are given by the function M. For transaction t, address a and value v (taken from a set V), we have

$$\begin{aligned}
M(\texttt{TXBegin}_t) &= \{\overline{\texttt{TXBegin}}_t\} \\
M(\texttt{TXCommit}_t) &= \{\overline{\texttt{TXCommit}}_t, \overline{\texttt{TXAbort}}_t\} \\
M(\texttt{TXWrite}_t(a, v)) &= \{\overline{\texttt{TXWrite}}_t, \overline{\texttt{TXAbort}}_t\} \\
M(\texttt{TXRead}_t(a)) &= \{\overline{\texttt{TXRead}}_t(v) \mid v \in V\} \cup \{\overline{\texttt{TXAbort}}_t\}
\end{aligned}$$

[3] In this paper, we use the word 'operation' in two senses. Here, we mean 'operation' as a component of the TM interface. Later, we use 'operation' to mean the instance of an operation within an execution. Both senses are standard, and any ambiguity is resolved by the context.

We let $\mathbf{TXBegin}_t$ denote the two-element sequence $\langle \texttt{TXBegin}_t, \overline{\texttt{TXBegin}_t} \rangle$, let $\mathbf{TXWrite}_t(x, v)$ denote $\langle \texttt{TXWrite}_t(x, v), \overline{\texttt{TXWrite}_t} \rangle$ and $\mathbf{TXRead}_t(x, v)$ denote $\langle \texttt{TXRead}_t(x), \overline{\texttt{TXRead}_t}(v) \rangle$, and finally let $\mathbf{TXCommit}_t$ denote $\langle \texttt{TXCommit}_t, \overline{\texttt{TXCommit}_t} \rangle$. We use notation '$\cdot$' for sequence concatenation.

Example 1. The following history is a possible execution of the TML, where the address x (initially 0) is accessed by two transactions 2 and 3 running concurrently.

$$\langle \texttt{TXBegin}_3, \texttt{TXBegin}_2, \overline{\texttt{TXBegin}_3}, \overline{\texttt{TXBegin}_2}, \texttt{TXWrite}_3(x, 4) \rangle \cdot \mathbf{TXRead}_2(x, 0) \cdot \langle \overline{\texttt{TXWrite}_3} \rangle \cdot \mathbf{TXCommit}_3$$

Note that operations overlap in this history. For example, the invocation $\texttt{TXBegin}_2$ appears between the invocation and response of transaction 3's $\texttt{TXBegin}$ operation. This overlapping means that this history represents an execution with both concurrent transactions, and concurrent operations. There is an important subset of histories, called *alternating histories* that do not have overlapping operations. That is, a history h is *alternating* if $h = \varepsilon$ (the empty sequence) or h is an alternating sequence of invocation and matching response events starting with an invocation and possibly ending with an invocation. Alternating histories represent executions in which the TM operations are atomic. Note that transactions may still be interleaved in an alternating history; only concurrency *between* operations is prohibited.

For a history $h = \langle h_1, h_2, \ldots h_n \rangle$, let $h|t$ be the projection onto the events of transaction t and $h[i..j]$ be the sub-sequence of h from h_i to h_j inclusive. We will assume that $h|t$ is alternating for any history h and transaction t. Note that this does not necessarily mean h is alternating itself. Opacity is defined for well-formed histories, which formalise the allowable interaction between a TM implementation and its clients. A projection $h|t$ of a history h onto a transaction t is *well-formed* iff it is ε or it is an alternating sequence of t-indexed invocations and matching responses, beginning with $\texttt{TXBegin}_t$, containing at most one each of $\texttt{TXBegin}_t$ and $\texttt{TXCommit}_t$, and containing no events after any $\overline{\texttt{TXCommit}_t}$ or $\overline{\texttt{TXAbort}_t}$ event. Furthermore, $h|t$ is *committed* whenever the last event is $\overline{\texttt{TXCommit}_t}$ and *aborted* whenever the last event is $\overline{\texttt{TXAbort}_t}$. In these cases, the transaction $h|t$ is *completed*, otherwise it is *live*. A history is *well-formed* iff $h|t$ is well-formed for every transaction t. The history in Example 1 is well-formed, and contains a committed transaction 3 and a live transaction 2.

Opacity. The basic principle behind the definition of opacity (and similar definitions) is the comparison of a given concurrent history against a sequential one. Opacity imposes a number of constraints, that can be categorised into three main types:

- *ordering constraints* that describe how events occurring in a concurrent history may be sequentialised;
- *semantic constraints* that describe validity of a sequential history hs; and
- a *prefix-closure constraint* that requires that each prefix of a concurrent history can be sequentialised so that the ordering and semantic constraints above are satisfied.

To help formalise these opacity constraints we introduce the following notation. We say a history h is *equivalent* to a history h', denoted $h \equiv h'$, iff $h|t = h'|t$ for all transactions $t \in T$. Further, the *real-time order* on transactions t and t' in a history h

is defined as $t \prec_h t'$ if t is a completed transaction and the last event of t in h occurs before the first event of t'.

Sequential history semantics. We now formalise the notion of sequentiality for transactions, noting that the definitions must also cover live transactions. A well-formed history h is *non-interleaved* if transactions do not overlap. 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 the notion of a valid sequential behaviour on a single address. To this end, we model shared memory by a set A of addresses mapped to values denoted by a set V. Hence the type $A \to V$ describes the possible states of the shared memory.

Definition 1 (Valid history). *Let $h = \langle h_0, \ldots, h_{2n-1} \rangle$ be an alternating history ending with a response (recall that an alternating history is a sequence of alternating invocation and response events starting with an invocation). We say h is* valid *if there exists a sequence of states $\sigma_0, \ldots, \sigma_n$ such that $\sigma_0(a) = 0$ for all addresses a, and for all i such that $0 \leq i < n$ and $t \in T$:*

1. *if $h_{2i} = \mathtt{TXWrite}_t(a, v)$ and $h_{2i+1} = \overline{\mathtt{TXWrite}}_t$ then $\sigma_{i+1} = \sigma_i[a := v]$; and*
2. *if $h_{2i} = \mathtt{TXRead}_t(a)$ and $h_{2i+1} = \overline{\mathtt{TXRead}}_t(v)$ then both $\sigma_i(a) = v$ and $\sigma_{i+1} = \sigma_i$ hold; and*
3. *for all other pairs of events $\sigma_{i+1} = \sigma_i$.*

A correct TM must ensure that all reads are consistent with the writes of the executing transaction as well as all previously committed writes. On the other hand, writes of aborted transactions must not be visible to other transactions. We therefore define a notion of *legal* histories, which are non-interleaved histories where only the writes of successfully committed transactions are visible to subsequent transactions.

Definition 2 (Legal history). *Let hs be a non-interleaved history, i an index of hs, and 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. 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 additionally lift to the level of specifications.

Definition 3 (Sequential history). *A well-formed history hs is* sequential *if it is non-interleaved and legal. We let $\mathcal{S}$ denote the set of all well-formed sequential histories.*

Transactional history semantics. 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 commit-pending transaction may already be visible to other transactions. To help account for this possibility, we must complete histories by (i) extending a history by adding responses to pending operations, then (ii) removing any pending operations that are left over. For (i), for each history h, we define a set $extend(h)$ that contains all histories obtained by adding to h response events matching any subset of the pending invocations in h. For (ii), for a history h, we let $[h]$ denote the history h with all pending invocations removed.

Definition 4 (Opaque history, Opaque object). *A history h is* final-state opaque *iff for some $he \in extend(h)$, there exists a sequential history $hs \in \mathcal{S}$ such that $[he] \equiv hs$ and furthermore $\prec_{[he]} \subseteq \prec_{hs}$. A history h is* opaque *iff each prefix h' of h is final-state opaque; a set of histories $\mathcal{H}$ is* opaque *iff each $h \in \mathcal{H}$ is opaque; and a TM implementation is* opaque *iff its set of histories is opaque.*

In Definition 4, conditions $[he] \equiv hs$ and $\prec_{[he]} \subseteq \prec_{hs}$ 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 final-state opacity is checked for each prefix of $[he]$.

Example 2. The history in Example 1 is opaque; a corresponding sequential history is

$$\textbf{TXBegin}_2 \cdot \textbf{TXRead}_2(x, 0) \cdot \textbf{TXBegin}_3 \cdot \textbf{TXWrite}_3(x, 4) \cdot \textbf{TXCommit}_3$$

Note that reordering of $\texttt{TXRead}_2(x, 0)$ and $\texttt{TXBegin}_3$ is allowed because their corresponding transactions overlap (even though the operations themselves do not).

3 Proving opacity via linearizability

In this section, we describe our method in detail, and we illustrate it by showing how to verify the simple TML algorithm presented in Section 2. Briefly, our method proceeds as follows.

1. Given a TM implementation, we construct a *coarse-grained abstraction* (CGA). This intermediate abstraction supports the standard transactional operations (begin, read, write and commit), and the effect of each operation is atomic. The states of this abstraction are simplified versions of the states of the original implementation, since the variables that are used for fine-grained synchronisation can be removed.
2. We prove that this CGA is opaque. The coarse-grained atomicity and simplifed state space of this abstraction mean that this opacity proof is much simpler than the direct opacity proof of the original implementation. Importantly, we do not need to consider the fine-grained synchronisation mechanisms of the original implementation in this part of the proof.
3. We prove that the original TM implementation is linearizable with respect to the CGA. Only at this point is it necessary to consider the complex interleaving and fine-grained synchronization of the actual TM implementation. As we noted in the introduction, for this linearizability proof we can leverage the powerful techniques for verifying linearizability that have been developed in recent years.

Formally, we regard our TM implementations, and our CGAs as sets of histories (consistent with the definition of opacity). The histories of the TM implementation must model all possible behaviours of the algorithm, and therefore some of these histories may contain overlapping operations. However, because the operations of the CGA are atomic, all the histories of the CGA are alternating.

Because the histories of each CGA are alternating, it is possible to prove that the original TM implementation is linearizable with respect to the CGA. To show how this works, we briefly review the definition of linearizability [17]. As with opacity, the

formal definition of linearizability is given in terms of histories: for every concurrent history an equivalent alternating history must exist that preserves the real-time order of operations of the original history. The *real-time order* on operations[4] o_1 and o_2 in a history h is given by $o_1 \prec_h o_2$ if the response of o_1 precedes the invocation of o_2 in h.

As with opacity, the given concurrent history may be incomplete, and hence, may need to be extended using $extend$ and all remaining pending invocations may need to be removed. We say $lin(h, ha)$ holds iff both $[h] \equiv ha$ and $\prec_{[h]} \subseteq \prec_{ha}$ hold.

Definition 5 (Linearizability). *A history h is* linearized *by alternating history ha iff there exists a history $he \in extend(h)$ such that $lin(he, ha)$. A concurrent object is linearizable with respect to a set of alternating histories $\mathcal{A}$ (in our case a CGA) if for each concurrent history h, there is a history $ha \in \mathcal{A}$ that linearizes h.*

In the remainder of this section, we flesh out our technique by verifying the TML algorithm presented in Section 2.

A coarse-grained abstraction Pseudocode describing the coarse-grained abstraction that we use to prove opacity of the TML is given in Listing 2. Like TML in Listing 1, it uses meta-variables loc_t (local to transaction t) and glb (shared by all transactions). Each operation is however, significantly simpler than the TML operations, and performs the entire operation in a single atomic step. The code for each operation is defined by wrapping the original code in an atomic block. However, the atomicity of the resulting method means that further simplifications can be made. For example, in the `TXRead` operation, the local variable v_t is no longer needed, and so can be removed. Likewise, CAS of the `TXWrite` operation is no longer required, and can also be dropped.

This basic strategy of making each operation atomic and then simplifying away any unnecessary state is sufficient for the examples we have considered. Indeed, when we apply our technique to the substantially more complicated NoRec algorithm, we find that the simplification step removes a great deal of complexity, including the entirety of NoRec's transactional validation procedure (Section 5).

Finding a CGA for any given TM algorithm is generally straightforward. We can provide three simple steps, or heuristics, that can be applied to find a useful CGA for any transactional memory algorithm. (1) We make every operation atomic in a naive way, essentially by surrounding the code in atomic blocks. (2) Much of the complexity in a transactional memory algorithm is often fine-grained concurrency control, such as locking, ensuring that each operation remains linearizable. This fine grained concurrency control can be removed in the CGA. (3) Concurrent/linearizable data structures in the implementation of the algorithm can be replaced by simple abstractions, that need not be implementable. For example, in the NORec algorithm (see Section 5) the write set and read sets are replaced with ordinary sets, and the validation routine becomes a predicate over these sets.

Opacity of the coarse-grained abstraction We turn now to the question of proving that our CGA is opaque. While our TM implementations and CGAs are sets of histories, it is convenient to define these models operationally using labelled transition systems

[4] Note: this differs from the real-time order on transactions defined in Section 2

Listing 2 TML-CGA: Coarse-grained abstraction of TML

```
 1: procedure INIT
 2:     glb ← 0
 3: procedure ATXBegin_t
 4:     atomic
 5:         await even(glb)
 6:         loc_t ← glb
 7: procedure ATXCommit_t
 8:     atomic
 9:         if odd(loc_t) then
10:             glb++
11: procedure ATXRead_t(a)
12:     atomic
13:         if glb = loc_t then
14:             return mem(a)
15:         else abort
16: procedure ATXWrite_t(a, v)
17:     atomic
18:         if glb ≠ loc_t then
19:             abort
20:         if even(loc_t) then
21:             loc_t++; glb++
22:         mem(a) ← v
```

that generate the appropriate sets of histories (so that the labels of the transition systems are invocation or response events). We do this for two reasons. First, the algorithms of interest work by manipulating state, and these manipulations can be mapped directly to labelled transition systems. The second reason relates to how we we prove that our CGAs are opaque.

We prove that our CGAs are opaque using techniques described in [11]. This means we leverage two existing results from the literature: the TMS2 specification by Doherty *et al.* [11], and the mechanised proof that TMS2 is opaque by Lesani *et al.* [21]. Using these results, it is sufficient that we prove trace refinement (i.e., trace inclusion of visible behaviour) between TML-CGA and the TMS2 specification. The rigorous nature of these existing results means that a mechanised proof of refinement against TMS2 also comprises a rigorous proof of opacity of TML-CGA.

Although TMS2 simplifies proofs of opacity, using it to verify an implementation still involves a complex simulation argument [20]. On the other hand, using TMS2 to prove opacity of a coarse-grained abstraction (CGA) is simple: the operations of the CGA are atomic, and hence, each of its operations corresponds exactly one operation of TMS2. This one-one correspondence also makes the invariants and simulation relations needed for the proof straightforward to establish. There are at most four main proof steps to consider, corresponding to the main steps of the TMS2 specification.

Theorem 1. TML-CGA *is opaque.*

Linearizability against the coarse-grained abstraction Having established opacity of TML-CGA, we can now focus on linearizability between TML and TML-CGA, which by Theorem 2 will ensure opacity of TML. As with the opacity part, we are free to use any of the available methods from the literature to prove linearizability [12]. We opt for a model-checking approach; part of our motivation is to show that model checking indeed becomes a feasible technique for verifying opacity.

We use the PAT model checker [26], which enables one to verify trace refinement (in a manner that guarantees linearizability) without having to explicitly define invariants, refinement relations, or linearization points of the algorithm. Interestingly, the model checker additionally shows that, for the bounds tested, TML is *equivalent* to TML-CGA, i.e., both produce exactly the same set of observable traces (see Lemma 1 below).

PAT allows one to specify algorithms using a CSP-style syntax [18]. However, in contrast to conventional CSP, events in PAT are arbitrary programs assumed to execute

atomically — as such they can directly modify shared state, and do not have to communicate via channels with input/output events. This enables our transactional memory algorithms to be implemented naturally. We obtain the following lemma, where constant $SIZE$ denotes the size of the memory (i.e., number of addresses) and constant V for the possible values in these addresses.

Lemma 1. *For bounds $N = 3$, $SIZE = 4$, and $V = \{0, 1, 2, 3\}$, as well as $N = 4$, $SIZE = 2$, and $V = \{0, 1\}$,* TML *is equivalent to* TML-CGA.

4 Soundness and completeness

We now present two key theorems for our proof method. Theorem 2, presented below, establishes soundness. That is, it states if we have an opaque CGA $\mathcal{A}$ (expressed as a set of alternating histories), and a TM implementation $\mathcal{H}$ (expressed as a set of concurrent histories) such that every history in $\mathcal{H}$ is linearizable to a history in $\mathcal{A}$, then every history in $\mathcal{H}$ is opaque. We prove Theorem 2 using the following lemma, which essentially states our soundness result for individual histories, rather than sets of histories.

Lemma 2 (Soundness per history [9]). *Suppose h is a concrete history. For any alternating history ha that linearizes h, if ha is opaque then h is also opaque.*

The main soundness theorem lifts this result to sets of histories. Its proof follows from Lemma 2 in a straightforward manner (see [2] for details).

Theorem 2 (Soundness). *Suppose $\mathcal{A}$ is a set of alternating opaque histories. Then a set of histories $\mathcal{H}$ is opaque if for each $h \in \mathcal{H}$, there exists a history $ha \in \mathcal{A}$ and an $he \in extend(h)$ such that $lin(he, ha)$.*

The next two results establish completeness of our proof method. Theorem 3 states that given an opaque TM implementation $\mathcal{H}$ (expressed as a set of concurrent histories) we can find a set of alternating opaque histories $\mathcal{A}$ such that every history in $\mathcal{H}$ can be linearized to a history in $\mathcal{A}$. Here, $\mathcal{A}$ is the CGA of our method. We prove this theorem using Lemma 3, which essentially states our completeness result for individual histories.

Lemma 3 (Existence of linearization). *If h is an opaque history then there exists an alternating history ha such that $lin(h, ha)$ and ha is final-state opaque.*

Proof. From the assumption that h is opaque, there exists an extension $he \in extend(h)$ and a history $hs \in \mathcal{S}$ such that $[he] \equiv hs$ and $\prec_{[he]} \subseteq \prec_{hs}$. Our proof proceeds by transposing operations in hs to obtain an alternating history ha such that $lin(he, ha)$. Our transpositions preserve final-state opacity, hence ha is final-state opaque.

We consider pairs of operations o_t and $o_{t'}$ such that $o_t \prec\!\!\!\prec_{hs} o_{t'}$, but $o_{t'} \prec\!\!\!\prec_{[he]} o_t$, which we call *mis-ordered pairs*. If there are no mis-ordered pairs, then $lin(he, hs)$, and we are done. Let o_t and $o_{t'}$ be the mis-ordered pair such that the distance between o_t and $o_{t'}$ in hs is least among all mis-ordered pairs. Now, hs has the form $\ldots o_t g o_{t'} \ldots$. Note that g does not contain any operations of transaction t, since if there were some operation o of t in g, then because opacity preserves program order and $o_t \prec\!\!\!\prec_{hs} o$, we

would have $o_t \prec_{[he]} o$. Thus $o, o_{t'}$ would form a mis-ordered pair of lower distance, contrary to hypothesis. For a similar reason, g does not contain any operations of t'. Thus, as long as we do not create a new edge in the opacity order $\prec_{hs}$, we can reorder hs to (1) $\ldots go_{t'}o_t \ldots$ or (2) $\ldots o_{t'}o_t g \ldots$ while preserving opacity. A new edge can be created only by reordering a pair of begin and commit operations so that the commit precedes the begin. If o_t is not a begin operation, then we choose option (1). Otherwise, note that $o_{t'}$ cannot be a commit, because since $o_{t'} \prec_{[he]} o_t$, $t' \prec t$, and thus t could not have been serialised before t'. Since $o_{t'}$ is not a commit, we can choose option (2). Finally, we show that the new history has no new mis-ordered pairs. Assume we took option (1). Then if there is some o in g such that $o_t \prec_{[he]} o$ we would have $o_{t'} \prec_{[he]} o$, and thus $o, o_{t'}$ would form a narrower mis-ordered pair. The argument for case (2) is symmetric. Thus, we can repeat this reordering process and eventually arrive at a final-state opaque history ha that has no mis-ordered pairs, and thus $lin(he, ha)$. □

Theorem 3 (Completeness). *If $\mathcal{H}$ is a prefix-closed set of opaque histories, then there is some prefix-closed set of opaque alternating histories $\mathcal{A}$ such that for each $h \in \mathcal{H}$ there is some $h' \in \mathcal{A}$ such that $lin(h, ha)$.*

Proof. Let $\mathcal{A} = \{h'.h' \text{ is final-state opaque and } \exists h \in \mathcal{H}.lin(h, h')\}$. Note that both the set of all opaque histories and the set of linearizable histories of any prefix-closed set are themselves prefix-closed. Thus, $\mathcal{A}$ is prefix closed. Because $\mathcal{A}$ is prefix-closed, and each element is final-state opaque, each element of $\mathcal{A}$ is opaque. For any $h \in \mathcal{H}$, Lemma 3 implies that there is some $ha \in \mathcal{A}$ that linearizes h. □

Note that the proof of Theorem 3 works by constructing the CGA $\mathcal{A}$ as a set of alternating histories. To construct the operational model that generates this set, we use the heuristics described in Section 3.

5 The NORec algorithm

In this section, we show that the method scales to more complex algorithms. In particular, we verify the NORec algorithm by Dalessandro *et al.* [8] (see Listing 3), a popular and performant software TM.

The verification for NORec proceeds as with TML. Namely, we construct a coarse-grained abstraction, NORec-CGA (see Listing 4), verify that NORec-CGA is opaque, then show that NORec linearizes to NORec-CGA. As with TML, we do not perform a full verification of linearizability, but rather, model check the linearizability part of the proof using PAT. The proof that NORec-CGA is opaque proceeds via forward simulation against a variant of TMS2 (TMS3), which does not require read-only transactions to validate during their commit, matching the behaviour of NORec more closely. We have proved (in Isabelle) that despite this weaker precondition for read-only commits, TMS2 and TMS3 are equivalent by proving each refines the other. Further details of TMS3 and proofs (including mechanisation) may be found in our extended paper [2]. The following theorem (proved in Isabelle) establishes opacity of NORec-CGA.

Theorem 4. NORec-CGA *is opaque.*

Listing 3 NORec pseudocode

```
1: procedure TXBegin_t
2:     do loc_t ← glb
3:     until even(loc_t)
4: procedure Validate_t
5:     while true do
6:         time_t ← glb
7:         if odd(time_t) then goto 6
8:         for a ↦ v ∈ rdSet_t do
9:             if mem(a) ≠ v then abort
10:        if time_t = glb then return time_t
11: procedure TXCommit_t
12:    if wrSet_t = ∅ then return
13:    while !cas(glb, loc_t, loc_t + 1) do
14:        loc_t ← Validate_t
15:    for a ↦ v ∈ wrSet_t do
16:        mem(a) ← v
17:    glb ← loc_t + 2
21: procedure TXWrite_t(a, v)
22:    wrSet_t ← wrSet_t ⊕ {a ↦ v}
23: procedure TXRead_t(a)
24:    if a ∈ dom(wrSet_t) then
25:        return wrSet_t(a)
26:    v_t ← mem(a)
27:    while loc_t ≠ glb do
28:        loc_t ← Validate_t
29:        v_t ← mem(a)
30:    rdSet_t ← rdSet_t ⊕ {a ↦ v_t}
31:    return v_t
```

Listing 4 NORec-CGA: Coarse-grained abstraction of NORec

```
1: procedure ATXBegin_t
2:     return
3: procedure ATXCommit(t)
4:     atomic
5:         if wrSet_t = ∅ then return
6:         else if rdSet_t ⊆ mem then
7:             mem ← mem ⊕ wrSet_t
8:         else abort
9: procedure ATXWrite_t(a, v)
10:    wrSet_t ← wrSet_t ⊕ {a ↦ v}
11: procedure ATXRead_t(a)
12:    atomic
13:        if a ∈ dom(wrSet_t) then
14:            return wrSet_t(a)
15:        else if rdSet_t ⊆ mem then
16:            rdSet_t ← rdSet_t ⊕ {a ↦ v}
17:            return mem(a)
18:        else abort
```

Next, we have a lemma that is proved via model checking [2].

Lemma 4. *For bounds $N = 2$, $SIZE = 2$ and $V = \{0, 1\}$,* NORec *is equivalent to* NORec-CGA.

TMS3 and NORec-CGA are similar in many respects. They both use read and write sets in the same way, and write-back lazily during the commit. The only additional information needed in the simulation is keeping track of the number of successful commits in NORec-CGA. Thus, the simulation relation used in the proof of Theorem 4 above is straightforward (see [2]). On the other hand, proving opacity of the fine-grained NoRec implementation directly would be much more difficult as we would need to concern ourselves with the locking mechanism employed during the commit to guarantee that the write-back occurs atomically. However, this locking mechanism is effectively only being used to guarantee linearizability of the NORec commit operation, so it need not occur in the opacity proof. Lesani *et al.* have directly verified opacity of

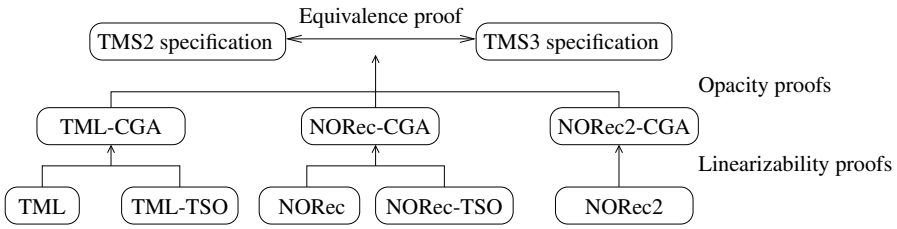

Fig. 1. Overview of proofs

NORec [20]. In comparison to our approach, they introduce several layers of intermediate automata, with each layer introducing additional complexity and design elements of the NORec algorithm. Overall, their proofs are much more involved than ours.

6 Weak Memory Models

We now demonstrate that our method naturally extends to reasoning about opacity of TM implementations under weak memory. We will focus on TSO in this Section, but our arguments and methods could be extended to other memory models. Note that we cannot employ a data-race freedom argument [1] to show that TML or NORec running on TSO are equivalent to sequentially consistent versions of the algorithms. This is because transactional reads can race with the writes of committing transactions (this is true even when we consider the weaker *triangular-race freedom* condition of [24]). This racy behaviour is typical for software transactional memory implementations.

There are two possibilities for verifying our TM algorithms on TSO. (1) Leveraging a proof of opacity of the implementation under sequential consistency then showing that the weak memory implementation refines this sequentially consistent implementation. (2) Showing that the implementation under weak memory linearizes to the coarse-grained abstraction directly. This approach simply treats an implementation executing under a particular memory model as an alternative implementation of the CGA algorithm in question.

In this paper, we follow the second approach, which shows that model checking linearizability of TSO implementations against a coarse-grained abstraction is indeed feasible. We verify both TML and NORec under TSO within the PAT model checker.

Due to the transitivity of trace inclusion, the proof proceeds by showing that the concrete implementation that executes using relaxed memory semantics linearizes to its corresponding coarse-grained abstraction. We use constant $BSIZE$ to bound the maximum size of the local buffer for each transaction.

Lemma 5. *For bounds* $N = 2$*,* $SIZE = 2$*,* $BSIZE = 2$ *and* $V = \{0, 1\}$*,* TML *under TSO is equivalent to* TML-CGA *and* NORec *under TSO is equivalent to* NORec-CGA*.*

7 Conclusions

Our main contributions for this paper are as follows. **(1)** We have developed a complete method for proving opacity of TM algorithms using linearizability. This allows one to

Listing 5 Abstraction used to verify TML in [9] is not opaque

```
1: procedure Begin_t
2:     return
3: procedure Write_t(a, v)
4:     atomic
5:         return abort or mem(a) ← v
6: procedure Commit_t
7:     return
8: procedure Read_t(a)
9:     atomic
10:        return mem(a) or return abort
```

reuse the vast literature on linearizability verification [12] (for correctness of the fine-grained implementation), as well as the growing literature on opacity verification (to verify the coarse-grained abstractions). **(2)** We have demonstrated our technique using the TML algorithm, and shown that the method extends to more complex algorithms by verifying the NORec algorithm. **(3)** We have developed an equivalent variation of the TMS2 specification, TMS3 that does not require validation when read-only transactions commit. Because TMS3 specifies equivalent behaviour to TMS2 while simplifying its preconditions, it is a preferable model for performing simulation proofs. **(4)** We have shown that the decomposition makes it possible to cope with relaxed memory by showing that both TML and NORec are opaque under TSO.

An overview of our proofs is given in Fig. 1. The equivalence proof between TMS2 and TMS3 as well as the opacity proofs between the CGAs and TMS2/3 specifications have been mechanised in Isabelle, whereas the linearizablity proofs are via model checking in PAT. We note that during our work, we developed a variation of NORec (called NORec2) which allows read operations to lookup values in the read set rather than querying shared memory, and demonstrated that this variation aborts less often than the existing NORec algorithm. We were able to quickly verify this modified algorithm. For more details, see [2].

Related work. Derrick *et al.* give a proof method that inductively generates a linearized *history* of a fine-grained implementation and shows that this linearized history is opaque [9]. Although checking opacity of a linearized history is simpler than a proof of opacity of the full concurrent history, one cannot consider their proof method to be a decomposition because the main invariant of the implementation must explicitly assert the existence of an opaque history (see Section 7). However, these methods suggest a crucial insight: that linearizability provides a sound method for proving opacity.

The basic idea of using a linearized history to verify opacity appears in [9], but their proof technique has little in common with ours. The abstraction that Derrick *et al.* use to motivate linearizability is given in Listing 5. Note that the read and commit operations in this abstraction perform no validation, and this abstraction is not opaque by itself. Therefore, it cannot be as a genuine intermediate specification. Instead, the steps of this abstraction are explicitly coupled with the *linearization points* of the fine-grained TML implementation, and it is this coupled system that is shown to be opaque. It is currently unclear if such a method could scale to more complex algorithms such as NORec, or to include weak memory algorithms.

Lesani and Palsberg have developed a technique that allows opacity to be checked by verifying an invariant-like condition called *markability* [22]. Lesani *et al.* have developed a second method [20] that proves opacity using the intermediate TMS2 specification [11, 21] using stepwise refinement against several intermediate layers of abstraction. Guerraoui *et al.* have developed an approach, where a set of *aspects* of an

algorithm are checked, followed by model checking over a *conflict-freedom* condition that implies opacity [13]. Koskinen and Parkinson [19] have a technique where they describe a push/pull model of transactions, and note that opaque transactions are a special case of push/pull transactions that do not pull during execution. This allows opacity to be proven via mapping the algorithm to the rules of the push/pull automata, which are stated in terms of commutativity conditions. In the context of our work, one could see such push/pull automata as an alternative to TMS2—one could use their proof technique to prove that our CGAs are opaque, and then use traditional linearizability verification techniques. As such, our work allows for an additional degree of proof decomposition. A key advantage of our method is that it is agnostic as to the exact techniques used for both the linearizability and opacity verifications, allowing for full verification by any method, or as in our case a mix of full verification and model-checking.

Experiences. Our experiences suggest that our techniques do indeed simplify proofs of opacity (and their mechanisation). Opacity of each coarse-grained abstraction is generally trivial to verify (our proofs are mechanised in Isabelle), leaving one with a proof of linearizability of an implementation against this abstraction. We emphasise that the method used for the second step is limited only by techniques for verifying linearizability. We have opted for a model checking approach using PAT, which enables linearizability to be checked via refinement. It is of course also possible to perform a full verification of linearizability. Furthermore, we note that we were able to use the model-checking approach to quickly test small variants of the existing algorithms.

Future work. Our work suggests that to fully verify a TM algorithm using coarse-grained abstraction, the bottleneck to verification is the proof of linearizability itself [12]. It is hence worthwhile considering whether linearizability proofs can be streamlined for transactional objects. For example, Bouajjani *et al.* have shown that for particular inductively-defined data structures, linearizability can be reduced to state reachability [4]. Exploration of whether such methods apply to transactional objects remains a topic for future work. Establishing this link would be a useful result — it would allow one to further reduce a proof of opacity to a proof of state reachability.

Acknowledgements. We thank John Derrick for helpful discussions and funding from EPSRC grant EP/N016661/1.

References

1. Adve, S.V., Aggarwal, J.K.: A unified formalization of four shared-memory models. IEEE Trans. Parallel Distrib. Syst. 4(6), 613–624 (Jun 1993)
2. Armstrong, A., Dongol, B., Doherty, S.: Reducing Opacity to Linearizability: A Sound and Complete Method. ArXiv e-prints (Oct 2016), https://arxiv.org/abs/1610.01004
3. Attiya, H., Gotsman, A., Hans, S., Rinetzky, N.: A programming language perspective on transactional memory consistency. In: Fatourou, P., Taubenfeld, G. (eds.) PODC '13. pp. 309–318. ACM (2013)
4. Bouajjani, A., Emmi, M., Enea, C., Hamza, J.: On reducing linearizability to state reachability. In: Halldórsson, M.M., Iwama, K., Kobayashi, N., Speckmann, B. (eds.) ICALP. LNCS, vol. 9135, pp. 95–107. Springer (2015)

5. Cerný, P., Radhakrishna, A., Zufferey, D., Chaudhuri, S., Alur, R.: Model checking of linearizability of concurrent list implementations. In: CAV. LNCS, vol. 6174, pp. 465–479. Springer (2010)
6. Chakraborty, S., Henzinger, T.A., Sezgin, A., Vafeiadis, V.: Aspect-oriented linearizability proofs. Logical Methods in Computer Science 11(1) (2015)
7. Dalessandro, L., Dice, D., Scott, M.L., Shavit, N., Spear, M.F.: Transactional mutex locks. In: D'Ambra, P., Guarracino, M.R., Talia, D. (eds.) Euro-Par (2). LNCS, vol. 6272, pp. 2–13. Springer (2010)
8. Dalessandro, L., Spear, M.F., Scott, M.L.: NORec: streamlining STM by abolishing ownership records. In: Govindarajan, R., Padua, D.A., Hall, M.W. (eds.) PPoPP. pp. 67–78. ACM (2010)
9. Derrick, J., Dongol, B., Schellhorn, G., Travkin, O., Wehrheim, H.: Verifying opacity of a transactional mutex lock. In: FM. LNCS, vol. 9109, pp. 161–177. Springer (2015)
10. Doherty, S., Groves, L., Luchangco, V., Moir, M.: Formal verification of a practical lock-free queue algorithm. In: FORTE. LNCS, vol. 3235, pp. 97–114. Springer (2004)
11. Doherty, S., Groves, L., Luchangco, V., Moir, M.: Towards formally specifying and verifying transactional memory. Formal Asp. Comput. 25(5), 769–799 (2013)
12. Dongol, B., Derrick, J.: Verifying linearisability: A comparative survey. ACM Comput. Surv. 48(2), 19 (2015)
13. Guerraoui, R., Henzinger, T.A., Singh, V.: Model checking transactional memories. Distributed Computing 22(3), 129–145 (2010)
14. Guerraoui, R., Kapalka, M.: On the correctness of transactional memory. In: Chatterjee, S., Scott, M.L. (eds.) PPOPP. pp. 175–184. ACM (2008)
15. Guerraoui, R., Kapalka, M.: Principles of Transactional Memory. Synthesis Lectures on Distributed Computing Theory, Morgan & Claypool Publishers (2010)
16. Harris, T., Larus, J.R., Rajwar, R.: Transactional Memory, 2nd edition. Synthesis Lectures on Computer Architecture, Morgan & Claypool Publishers (2010)
17. Herlihy, M., Wing, J.M.: Linearizability: A correctness condition for concurrent objects. ACM TOPLAS 12(3), 463–492 (1990)
18. Hoare, C.A.R.: Communicating sequential processes. Commun. ACM 21(8), 666–677 (1978)
19. Koskinen, E., Parkinson, M.: The push/pull model of transactions. In: PLDI. PLDI '15, vol. 50, pp. 186–195. ACM, New York, NY, USA (Jun 2015)
20. Lesani, M., Luchangco, V., Moir, M.: A framework for formally verifying software transactional memory algorithms. In: Koutny, M., Ulidowski, I. (eds.) CONCUR 2012. pp. 516–530. Springer Berlin Heidelberg (2012)
21. Lesani, M., Luchangco, V., Moir, M.: Putting opacity in its place. In: Workshop on the Theory of Transactional Memory (2012)
22. Lesani, M., Palsberg, J.: Decomposing opacity. In: Kuhn, F. (ed.) DISC. LNCS, vol. 8784, pp. 391–405. Springer (2014)
23. Liu, Y., Chen, W., Liu, Y.A., Sun, J., Zhang, S.J., Dong, J.S.: Verifying linearizability via optimized refinement checking. IEEE Trans. Software Eng. 39(7), 1018–1039 (2013)
24. Owens, S.: Reasoning about the Implementation of Concurrency Abstractions on x86-TSO. In: DHondt, T. (ed.) ECOOP 2010, LNCS, vol. 6183, pp. 478–503. Springer Berlin Heidelberg (2010)
25. Schellhorn, G., Derrick, J., Wehrheim, H.: A sound and complete proof technique for linearizability of concurrent data structures. ACM TOCL 15(4), 31:1–31:37 (2014)
26. Sun, J., Liu, Y., Dong, J.S., Pang, J.: PAT: towards flexible verification under fairness. In: CAV. LNCS, vol. 5643, pp. 709–714. Springer (2009)
27. Vafeiadis, V.: Modular fine-grained concurrency verification. Ph.D. thesis, University of Cambridge (2007)