Verifying Opacity of a Transactional Mutex Lock

John Derrick[1(✉)], Brijesh Dongol[2], Gerhard Schellhorn[3], Oleg Travkin[4], and Heike Wehrheim[4]

[1] Department of Computing, University of Sheffield, Sheffield, UK
j.derrick@dcs.shef.ac.uk
[2] Department of Computer Science, Brunel University, London, UK
[3] Universität Augsburg, Institut für Informatik, 86135, Augsburg, Germany
[4] Universität Paderborn, Institut für Informatik, 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 an atomic block. This atomicity property is captured by a correctness criterion called *opacity*. Opacity relates histories of a sequential atomic specification with that of STM implementations.

In this paper we prove opacity of a recently proposed STM implementation (a Transactional Mutex Lock) by Dalessandro et al.. The proof is carried out within the interactive verifier KIV and proceeds via the construction of an intermediate level in between sequential specification and implementation, leveraging existing proof techniques for linearizability.

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 STMs is with database transactions, which perform a series of updates to data atomically in an all-or-nothing manner. If a transaction succeeds, all its operations succeed, and otherwise, all its operations fail. Since the first proposal of an STM [20], a number of STM implementations have been presented (e.g. [11,3]). 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. The locking mechanism of Transactional Mutex Locks [4] which we study in this paper implements an optimistic locking scheme. These currently find their way into standard programming languages, for instance via the new class `StampedLock` of the Java 8 release.

As STM implementations allow several operations to execute simultaneously, what one means by "correctness" is open to interpretation. Several notions of correctness have been defined, e.g., strict serializability [17], opacity [8,2], TMS1 and TMS2 [6], and virtual world consistency [13]. A number of researchers have already considered methods for verifying correctness of transactional memory implementations; a comprehensive survey may be found in [14]. Formal verification is clearly needed as STM

N. Bjørner and F. de Boer (Eds.): FM 2015, LNCS 9109, pp. 161–177, 2015.
DOI: 10.1007/978-3-319-19249-9_11

implementations employ fine-grained operations allowing interleavings between concurrent transactions, and subtle errors are therefore likely to arise but difficult to detect via e.g. testing.

In this paper, we provide the first formal, mechanised proof of correctness of the Transactional Mutex Lock (TML). As correctness criterion we employ the recently given definition of opacity of Attiya et al. [2]. 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 proof technique is fully mechanised within the interactive prover KIV [18] and leverages existing proof techniques [5] for linearizability [12].

More specifically, our approach consists of two steps: we (1) show that all runs of TML are linearizable to runs in which first of all reads and writes to memory occur atomically, and (2) establish an invariant about such runs stating that they all have "matching" runs 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).

The paper is structured as follows: Section 2 gives an introduction to software transactional memory, presents our case study and defines the correctness criterion of opacity. Our general proof approach with steps (1) and (2) is described in Section 3; Section 4 explains both steps for our case study, the Transactional Mutex Lock. Section 5 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) [10]. 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.

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`)[1]. 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 starting 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.

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

```
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++;  // LP
B3 return ok;                     E3 return commit;

TMRead(addr):                     TMWrite(addr,val):
R1 tmp := *addr;                  W1 if (loc & 0)
R2 if (glb = loc) // LP           // W2 is LP when glb ≠ loc
R3   return tmp;                  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)

2.1 Example: Transactional Mutex Lock

In this paper, we will study a particular implementation of STM, namely the Transactional Mutex Lock (TML) of Dalessandro *et al.* [4]. It provides exactly the four types of operations, but operation TMEnd will never respond with abort. See Fig. 1 (the references in the comments to LP are explained later).

The TML uses a global counter *glb* (initially 0) shared by all processes, and local variables *tmp* (temporarily storing the value read from an address) and *loc* (storing a copy of *glb*). Variable *glb* records whether there is a live writing transaction. Namely, *glb* is odd if there is a live writing transaction, and even otherwise. Initially, *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 1. 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 incremented (to an even value), allowing other transactions to begin.

Table 1. Events appearing in the histories of TML

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

The key question we want to answer in this paper is: "Does the TML correctly implement an STM", i.e., 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 a "correctness" for an STM implementation as *opacity* [8]. We formalise this via a series of definitions leading up to the definition of an opaque history in Definition 5 below.

2.2 Opacity

There are numerous formalizations of opacity in the literature; our definition mainly follows Attiya et al. [2]. We model shared memory by a set *Addr* of addresses or locations. For simplicity we assume addresses hold integer, denoted $\mathbb{Z}$, values only, hence $State == Addr \to \mathbb{Z}$ describes the possible states of the shared memory. Initially, all addresses hold the value 0. 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, the 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.

Example 1. 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.

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

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

Histories. We're 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're 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 we use an intermediate specification which has *alternating histories*, which we define now.

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 *finishing*, otherwise t is *live*. A history is *well-formed* if it consists of transactions only and at most one live transaction per process.

Example 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. □

The basic principle behind the definition of opacity (and similar definitions) 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.

Sequential histories. 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 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 $h = h_1 \frown \langle inv_p(\texttt{TMBegin})\rangle \frown h_2 \frown \langle inv_q(\texttt{TMBegin})\rangle \frown h_3$ 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 (Valid history). *Let $h = \langle ev_0, \ldots, ev_{2n-1}\rangle$ be a sequence of alternating invocation and response events 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 $\sigma_0(x) = 0$ for all $x \in Addr$ and, for all i such that $0 \leq i < n$ and $p \in P$:*

1. *if* $ev_{2i} = inv_p(\texttt{TMWrite}(x, v))$ *and* $ev_{2i+1} = res_p(\texttt{TMWrite}(\texttt{ok}))$ *then* $\sigma_{i+1} = \sigma_i[x := v]$
2. *if* $ev_{2i} = inv_p(\texttt{TMRead}(x))$ *and* $ev_{2i+1} = res_p(\texttt{TMRead}(v))$ *then* $\sigma_i(x) = v$ *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 when the transaction commits. Writes in a transaction that abort don't effect 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 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 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. Opacity relates concurrent histories that an implementation generates to sequential histories. We say a history h is *equivalent* to a history h', 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 finished transaction and the last event of t_1 in h occurs before the first event of t_2.

A given concrete history may be incomplete, i.e., consist of pending operation calls, which may be distinguished in a history as an invocation that has no matching response. As some of these pending calls may have taken effect, pending operation calls may be completed by adding matching responses. There may also be incomplete operation calls that have not taken effect; it is safe to remove the pending invocations. It is however not possible to determine whether or not a pending operation call has taken effect from the history only; therefore, we define a function $complete(h)$ that constructs all possible completions of h by appending matching responses and removing pending invocations.

Definition 5 (Opaque history). *A history h is* 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 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.* □

Example 3. The above history h_1 is opaque; the corresponding sequential history is

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

However, a history may not be opaque for several reasons. A very simple example is h_2, which violates memory semantics, since it reads a value 4, that has not been written:

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

A second more complex example is h_3.

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

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

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

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 runs first (and reads $x = y = 0$), or transaction 2 runs first (in which case transaction 1 should read $x = y = 4$). The TML will prevent h_4 — the second read of transaction 1 will abort because its *loc* value is smaller than *glb*, which was incremented by the first write of transaction 2. However, in general an implementation could allow transaction 2 to read $y = 0$, i.e., if we replace the last event in h_4 by $res_2(\texttt{TMRead}(0))$, the modified history is still opaque.

Thus our question of implementation correctness of the TML 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 general proof method for showing opacity of TML.

Aside. Neither h_3 nor h_4 violate strict serializability [17]. 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 reading and writing transaction variables is 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

```
ABegin:                    AEnd:
  return ok                  return commit

ARead(addr):               AWrite(addr,val):
  atomic {                   atomic {
    return addr                addr := val ; return ok
    or                         or
    return abort }             return abort }
```

Fig. 2. Atomic specification of an STM

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.

3 A Proof Method for Opacity

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 unbounded (but finite) sequences of transactional memory operations are considered atomic. The operations of STM implementations are however simple: there are operations to begin and end a transaction and operations to read and write from memory. The majority of these leave memory unchanged; for our TML example, the only operation that modifies memory is a write operation that does not abort. Note that this is not the only possibility — there are STM implementations that use *deferred updates*, where write operations leave memory unchanged and writes are only performed when transactions end.

Our proof method uses an intermediate specification which is an *atomic specification* of an STM implementation (with non deferred updates) where each operation is atomic, thus interleaving of statements within an operation does not occur.

The proof method works by (a) showing that every history in the TML implementation can be linearized by an alternating history of this intermediate specification, and (b) these alternating histories are themselves opaque. We describe the proof method using TML as a running example. Our proofs have been fully automated in KIV [18], the resulting development may be viewed online(https://swt.informatik.uni-augsburg.de/swt/projects/TML.html). The link also contains additional notes on our KIV proof.

3.1 Defining an Atomic Specification of an STM

The definition of the intermediate specification is simple, and the atomic specification of an STM is given in Fig. 2. For example, the `ARead(x)` operation is an abstraction of `TMRead(x)` that reads and returns the value of x in a single atomic step or it aborts. (We assume that `or` defines a non-deterministic choice.)

3.2 Linearizability and Opacity

Correctness of the TML implementation is shown using *linearizability* [12], which is the standard correctness criterion for concurrent objects. The idea of linearizability is:

> 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 they may take effect in any order from an abstract perspective, but otherwise they must take effect in the order in which they are invoked. This provides a meaning for fine-grained concurrent objects with overlapped operation calls in terms of the abstract object, whose operation calls do not overlap.

As with opacity, the formal definition of linearizability is given in terms of histories (of invocation/response events); for every concurrent history we have to find an equivalent alternating (invocations immediately followed by the matching response) history that preserves real time order of operations. The *real-time order* on operation calls[2] o_1 and o_2 in a history h is defined as $o_1 \prec_h o_2$ 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 pending invocations to a history h, then removes any remaining pending invocations.

Definition 6 (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 $\prec_{hc} \subseteq \prec_{ha}$. A concurrent object is linearizable with respect to a specification if for each concurrent history h, there is an alternating history ha of the specification 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 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 $\prec_{hc} \subseteq \prec_{ha}$ 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 $\prec_{hc} \subseteq \prec_{ha}$ 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. □

In applying this to TML, we show that every concurrent history h will be linearized by an alternating history ha of the intermediate specification given in Figure 2, and that every such ha is opaque.

Because the histories of the atomic specification of an STM are alternating, i.e., each operation invocation is immediately followed by its response, we further simplify reasoning by reasoning about *runs*, which abstractly represent alternating histories. Thus we specifically show that the run r corresponding to ha is opaque.

A *run* is a sequence of run events (see column 1 of Table 2), representing a matching invocation/response event pair; $Begin(p)$ denotes a `TMBegin` operation by process p; run

[2] Note: this is different from the real time order on transactions defined in Section 2.2

Table 2. Run events abstracting matching invocation/return pairs

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

events $Read(p,x,v)$ and *Write(p,x,v)* denote successful read and write operations by process p on address x with value v; run event $Commit(p)$ denotes a successful `TMBegin` operation by process p; and $Abort(p)$ denotes an operation invocation that aborts.

Example 4. The run corresponding to the history

$$\begin{aligned} ha \mathrel{\widehat{=}} \langle & inv_2(\texttt{TMBegin});\ res_2(\texttt{TMBegin(ok)});\ inv_2(\texttt{TMRead}(x);\ res_2(\texttt{TMRead}(0));\\ & inv_3(\texttt{TMBegin});\ res_3(\texttt{TMBegin(ok)});\ inv_3(\texttt{TMWrite}(x,4));\\ & res_3(\texttt{TMWrite(ok)});\ inv_3(\texttt{TMEnd});\ res_3(\texttt{TMEnd(commit)}),\\ & inv_2(\texttt{TMRead}(x);\ res_2(\texttt{TMRead(abort)})\rangle \end{aligned}$$

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

Because `Abort(p)` relates to several possible pairs, a run is more abstract than a history. Although it is possible to obtain a 1-1 correspondence between runs and histories by defining other types of run events, the encoding in this paper simplifies the mechanisation of the proof.

4 Proving Opacity of TML

In this section we apply the theory from the previous section and show how opacity of the TML may be proved. Section 4.1 describes how the TML may be modelled in KIV, Section 4.2 presents the linearizability proof and Section 4.3 the opacity of the runs recorded as part of this proof.

4.1 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 variable a and v storing the input/output addresses and values, respectively. We thus use the following state:

```
CState =
mkcs(. .glb : nat, . .mem : address → int, . .pc : Proc → PC,
     . .loc : Proc → nat, . .a : Proc → address, . .v : Proc → int)
```

Modelling atomic statements: Modelling an atomic statement of the TML as a KIV state transition is also straightforward; for example, consider 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, a, v, glb', mem', pc', loc', a', v')
↔
( pc = W2 ∧ loc' = loc ∧ mem' = mem ∧ a' = a ∧ v' = v
     ∧ glb' = (loc = glb ⊃ loc + 1;glb) ∧ pc' = (loc = glb ⊃ W3;W6) );
```

Promotion to system wide steps: Local specifications must now be promoted to the level of the system, where the system consists of the concrete state *cs* together with a variable *r* representing the run so far.

As promotion is a standard procedure [21], we omit the full details here. In KIV we define one generic promoted KIV state transition COp-def that gets instantiated to specific promoted transitions as necessary.

More interestingly, as part of the promotion we record run events in the run variable r, at the *linearization points* of the operations TMBegin, TMRead, TMEnd, and TMWrite.

The linearization points of these transitions are annotated in comments in the code in Figure 1. As with standard linearizability proofs, linearization points are often conditional and their locations sometimes not intuitive. An operation may either linearize the invoked operation or linearize to abort. Operation TMBegin linearizes at B1 if 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 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 successfully executed a TMWrite. If no writes were performed, then loc must be even; such a transaction 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 is E2.

The expression on the right of "r' =" below is an if-then-else expression describing the value of r' (i.e., the value of r in the post state). To save space some details are omitted, and replaced by "...". Thus, for example, the first condition states that r' is set to r + Begin(p), which concatenates Begin(p) to r, whenever pc = B1 $\wedge$ even(glb) holds in the pre-state, i.e., whenever process p executes line B1 where the value of glb is even.

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

4.2 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.

We thus show that h is linearized to a run r. This is done by proving two lemmas in KIV for each operation of transaction (TMWrite etc).

First, when executed by process p no operation ever passes more than one linearization point (LP) in any execution (regardless of other interleaved operations executed by other processes) before executing a return (so even nonterminating TMBegins never execute more than one LP).

Second, if the operation reaches a return and terminates, then it 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 $Write(p, x, v)$ to r when executing the instruction at W5 (and therefore actually writes v to $mem(x)$), and we prove that this is possible only when the input to the invoking instruction of TMWrite is x, v and the output is empty.

Note that this encoding is recording the (more abstract) runs directly, as opposed to recording an alternating history which is abstracted to runs as a separate step. This simplified the KIV proof significantly without affecting soundness. In particular, linearizability is guaranteed because the linearization points that occur are done by steps of the operations themselves (more intricate examples where linearization points are executed by other threads need more complex techniques, see [19] for a complete proof

method). The method used here is akin to the technique used in [22], where concrete states are augmented with auxiliary variables representing the abstract state together with additional modifications of the auxiliary state at the linearization points.

4.3 Step 2: Proving Opacity of Alternating Histories Using Runs

In this subsection we prove an alternating history which linearized 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 slight 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 Section 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 $mem \in State$ to produce the next state mem'. Notation $mem[x := v]$ denotes functional override, where $mem(x)$ is updated to v.

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

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 $\mathtt{r}$. 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, $[\![\mathtt{r}]\!](\sigma)$ 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 *interleaved* states that transactions may be overlap. The final conjuct ensures rs is both valid and legal as defined in Definitions 2 and 3, respectively, where *committed* restricts a given run to the committed runs plus the (live) transaction to which $r(n)$ belongs as defined in Definition 3.

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

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 obtained from ha by replacing each pair of matching events in ha by the corresponding run event from Table 2.

Theorem 2. *An alternating history ha is opaque if there exists a run r that corresponds to ha and r is opaque.*

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 execution of the TML augmented with runs (cs, r), it is possible to find an rs such that $opaque(r, rs)$.

As with our work on linearizability we prove this via construction of an appropriate invariant. The main proof then shows that all augmented states (cs, r) generated by a concurrent execution of the TML implementation satisfies the predicate $\exists rs \bullet INV(cs, r, rs)$. The formula $INV(cs, r, rs)$ defines a number of invariants for a sequential history rs, which in particular imply $opaque(r, rs)$, which we now explain.

The formula $INV(cs, r, rs)$ formalizes the observation that the (legal) transaction sequences rs generated by the TML implementation always consist of three parts: a first one 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. 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). 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. This implicitly implies that 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 $loc(p) = glb$ and $pc(p) = W5$, any finished transaction must occur before t.

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. Def. 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, we show that the invariant holds after any step of the TML specification that 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) is trivial.

Linearization steps of a TML operation add the corresponding run event re to r, i.e. $r' = r \frown \langle re \rangle$. The proof for the LP of `TMBegin` (i.e., line `B1`) is relatively simple, the new rs' has the newly started transaction concatenated at the end. For the other LPs, we use a function $tseq(rs)$, which generates a sequence of transactions from rs in order[3]. In particular, if $ts = tseq(rs)$, then $rs = ts(0) \frown ts(1) \frown \cdots \frown ts(\#ts - 1)$. At each LP, assuming $ts = tseq(rs)$, we add a run event re at the end of some $ts(j)$ and leave all other $ts(i)$ unchanged to generate a new ts'. The sequence ts' may also reorder transactions in ts that overlap in r', however, in most cases, the order of transactions is left unchanged, i.e. the choice for ts' is $ts[j := ts(j) \frown \langle re \rangle]$. We then consider $rs' = tseq(ts')$ and we show that this new rs' preserves the memory semantics.

Because opacity holds for the transaction sequence rs before the LP step, we know from Definition 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* $ts[0..k]$. 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 $last(\sigma)$.
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`. Both (a) and (b) may violate **INV3**, which is necessary to ensure that real-time order in r is preserved. For both scenarios, we must commute the transaction with current $loc(p)$ value. 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,

[3] 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.

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.

Proof statistics. 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 7LPs + the goal for pc=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 [3]) is a topic of future work.

5 Conclusions

There are many notions of correctness for STMs [14,9]. 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 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 1), while Theorem 2 justifies proving opacity of an alternating history by proving opacity of the run corresponding to the history.

Although there are several works comparing and contrasting different correctness conditions for STM (including opacity) (e.g., [6,16,1]), there only a handful of papers that consider verification of the STM implementations themselves. A model checking approach is presented in [7], however, the technique only considers *conflicts* between read and write operations in different transactions. More recently, Lesani has considered opacity verification of numerous algorithms [14], which includes techniques for reducing the problem of proving opacity into one of verifying a number of simpler invariants on the orders of events [15]. 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.

References

1. Attiya, H., Gotsman, A., Hans, S., Rinetzky, N.: Safety of live transactions in transactional memory: TMS is necessary and sufficient. In: Kuhn, F. (ed.) DISC 2014. LNCS, vol. 8784, pp. 376–390. Springer, Heidelberg (2014)
2. Attiya, H., Gotsman, A., Hans, S., Rinetzky, N.: A programming language perspective on transactional memory consistency. In: Fatourou, P., Taubenfeld, G. (eds.) PODC 2013, pp. 309–318. ACM (2013)
3. Dice, D., Shalev, O., Shavit, N.: Transactional locking II. In: Dolev, S. (ed.) DISC 2006. LNCS, vol. 4167, pp. 194–208. Springer, Heidelberg (2006)
4. Dalessandro, L., Dice, D., Scott, M.L., Shavit, N., Spear, M.F.: Transactional mutex locks. In: D'Ambra, P., Guarracino, M., Talia, D. (eds.) Euro-Par 2010, Part II. LNCS, vol. 6272, pp. 2–13. Springer, Heidelberg (2010)
5. Derrick, J., Schellhorn, G., Wehrheim, H.: Verifying linearisability with potential linearisation points. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 323–337. Springer, Heidelberg (2011)
6. Doherty, S., Groves, L., Luchangco, V., Moir, M.: Towards formally specifying and verifying transactional memory. Formal Asp. Comput. 25(5), 769–799 (2013)
7. Guerraoui, R., Henzinger, T.A., Singh, V.: Model checking transactional memories. Distributed Computing 22(3), 129–145 (2010)
8. Guerraoui, R., Kapalka, M.: On the correctness of transactional memory. In: Chatterjee, S., Scott, M.L. (eds.) PPOPP, pp. 175–184. ACM (2008)
9. Guerraoui, R., Kapalka, M.: Principles of Transactional Memory. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers (2010)
10. Harris, T., Larus, J.R., Rajwar, R.: Transactional Memory, 2nd edition. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers (2010)
11. Harris, T.L., Fraser, K.: Language support for lightweight transactions. In: Crocker, R., Steele Jr., G.L. (eds.) OOPSLA, pp. 388–402. ACM (2003)
12. Herlihy, M., Wing, J.M.: Linearizability: A correctness condition for concurrent objects. ACM TOPLAS 12(3), 463–492 (1990)
13. Imbs, D., Raynal, M.: Virtual world consistency: A condition for STM systems (with a versatile protocol with invisible read operations). Theor. Comput. Sci. 444, 113–127 (2012)
14. Lesani, M.: On the Correctness of Transactional Memory Algorithms. PhD thesis, UCLA (2014)
15. Lesani, M., Palsberg, J.: Decomposing opacity. In: Kuhn, F. (ed.) DISC 2014. LNCS, vol. 8784, pp. 391–405. Springer, Heidelberg (2014)
16. Luchangco, V., Lesani, M., Moir, M.: Putting opacity in its place. In: Workshop on the Theory of Transactional Memory (2012)
17. Papadimitriou, C.H.: The serializability of concurrent database updates. J. ACM 26(4), 631–653 (1979)
18. Reif, W., Schellhorn, G., Stenzel, K., Balser, M.: Structured specifications and interactive proofs with KIV. In: Automated Deduction—A Basis for Applications. Interactive Theorem Proving, vol. II, ch.1, pp. 13–39. Kluwer (1998)
19. Schellhorn, G., Derrick., J., Wehrheim, H.: A Sound and Complete Proof Technique for Linearizability of Concurrent Data Structures. ACM Trans. Comput. Logic, 15 (2014)
20. Shavit, N., Touitou, D.: Software transactional memory. Distributed Computing 10(2), 99–116 (1997)
21. Spivey, J.M.: The Z Notation: A Reference Manual. Prentice Hall (1992)
22. Vafeiadis, V.: Modular fine-grained concurrency verification. PhD thesis, University of Cambridge (2007)