Under consideration for publication in Formal Aspects of Computing

Towards Formally Specifying and Verifying Transactional Memory

Simon Doherty[1] and Lindsay Groves[1] and Victor Luchangco[2] and Mark Moir[2]

[1]School of Engineering and Computer Science, Victoria University of Wellington, New Zealand
[2]Oracle Labs, Burlington, MA, USA

Abstract. Over the last decade, great progress has been made in developing practical transactional memory (TM) implementations, but relatively little attention has been paid to precisely specifying what it means for them to be correct, or formally proving that they are.

In this paper, we present TMS1 (Transactional Memory Specification 1), a precise specification of correct behaviour of a TM runtime library. TMS1 targets TM runtimes used to implement transactional features in an unmanaged programming language such as C or C++. In such contexts, even transactions that ultimately abort must observe consistent states of memory; otherwise, unrecoverable errors such as divide-by-zero may occur before a transaction aborts, even in a correct program in which the error would not be possible if transactions were executed atomically.

We specify TMS1 precisely using an I/O automaton (IOA). This approach enables us to also model TM implementations using IOAs and to construct fully formal and machine-checked correctness proofs for them using well established proof techniques and tools.

We outline key requirements for a TM system. To avoid precluding any implementation that satisfies these requirements, we specify TMS1 to be as general as we can, consistent with these requirements. The cost of such generality is that the condition does not map closely to intuition about common TM implementation techniques, and thus it is difficult to prove that such implementations satisfy the condition.

To address this concern, we present TMS2, a more restrictive condition that more closely reflects intuition about common TM implementation techniques. We present a simulation proof that TMS2 implements TMS1, thus showing that to prove that an implementation satisfies TMS1, it suffices to prove that it satisfies TMS2. We have formalised and verified this proof using the PVS specification and verification system.

1. Introduction

Transactional memory (TM) [HM93] aims to make it significantly easier to develop and maintain concurrent programs that are scalable, efficient, and correct by allowing programmers to specify that a sequence of operations on shared objects should be executed as a *transaction*. A transaction makes the sequence of operations appear to be applied without interference from concurrent transactions, and without concurrent transactions observing partial results of the sequence. Application programmers are not concerned with *how* these guarantees are made; they simply identify sequences of operations that are to be treated as transactions, and enforcing the guarantees is the responsibility of the system.

Correspondence and offprint requests to: Victor Luchangco, Oracle Labs, Burlington, MA, USA, email: victor.luchangco@oracle.com

The power of transactions to allow programmers to write correct and efficient concurrent programs without being concerned with the details of how transactions are implemented has been clearly demonstrated for several decades in the context of database systems. TM aims to deliver similar benefits to shared memory programmers.

In typical implementations, concurrent transactions are executed in parallel, with the hope that they will not *conflict* (two transactions conflict if they access a data item in common, and one of them updates it). If a transaction encounters a conflict, either it or the transaction with which it conflicts can be *aborted*; none of the effects of an aborted transaction are visible to other transactions. Otherwise the transaction attempts to *commit*. If this attempt is successful, the effects of the transaction become visible to all other transactions instantaneously: no transaction ever observes partial results of another.

Because transactions aim to make it easier to develop correct concurrent programs, the correctness of their implementations is particularly important. We are therefore pursuing a long-term goal of developing formal and machine-checked correctness proofs for TM implementations. An important first step is to precisely specify what it means for a TM implementation to be correct, in a way that lends itself to rigourous proof. Despite significant progress towards practical TM implementations, there is not a well accepted correctness condition for them. Instead, a number of conditions have been proposed with varying properties and varying degrees of precision. As explained in Sections 4 and 7, we find these previous conditions unsatisfying for various reasons. In this paper, we present a new correctness condition TMS1 (Transactional Memory Specification 1; see Section 3), which lays a foundation for formally specifying and mechanically verifying correctness for existing and future TM implementations.

The reader may wonder why new correctness conditions for TM are needed when the use of transactions in database systems has a long, successful history, and correctness conditions have been propsed and extensively studied for that context (see Section 7). Indeed, to support the desired programming simplicity, TM must make guarantees similar to *serialisability* [Pap79], a long-established correctness condition for databases. Briefly, serialisability requires that, for any execution of the system, there is some total order (called a *serialisation*) over all successfully committed transactions in the execution such that the behaviour of the system is equivalent to behaviour in which the transactions are executed one at a time in this order. This allows programmers to think about transactions as if they are always executed serially, even though in reality they may execute concurrently. Similar properties are required of TM implementations.

However, depending on context, additional properties are necessary for TM implementations. Our new TM correctness condition TMS1 is intended to specify requirements for a TM runtime library in unmanaged languages such as C or C++, running on a system that may also support other mechanisms for coordinating threads. A key question to be answered by TMS1 is what guarantees are provided about the behaviour of transactions that abort, and of transactions that are active and may abort. In our target context, it is important that such transactions do not observe behaviour before they abort that is inconsistent with the programmer's mental model of transactions executing serially [DSS06, HSATH06]: a transaction that detects such inconsistency may cause unrecoverable errors, such as segmentation violations, divide-by-zero errors, etc. For example, suppose a program maintains an invariant that two variables x and y are never equal, and then divides by $x - y$ within a transaction. If that transaction reads values from x and y that are equal, it will cause a divide-by-zero error. That it will not commit successfully is little comfort in such cases!

In contrast, serialisability does not constrain the behaviour of transactions that abort at all, and therefore provides no protection against such problems. Furthermore, the order of committed transactions required by serialisability is *not* required to respect the real-time order of the transactions: two transactions that execute in series may be ordered in the opposite order for the purposes of serialisability. *Strict serialisability* [Pap79] refines serialisability by adding such a requirement. As argued in Section 3, this requirement is important for TM, and is therefore included in our TMS1 condition.

We do *not* consider TMS1 to be the definitive correctness condition for all TM runtimes. In particular, our target context (i.e., unmanaged languages) has significant influence on choices we make about the correctness condition, and different criteria apply for different contexts. Furthermore, even for this context, we have made certain simplifying assumptions for now. For example, we consider only the simple case in which variables are not shared between transactional and nontransactional code.

Nonetheless, we believe that TMS1 is an important contribution as a demonstration of how to specify TM correctness conditions in a precise way that facilitates formal, mechanical correctness proofs using well established proof techniques and tools. Specifically, we are following an approach that we have used successfully to verify several concurrent algorithms [DGLM04, CDG05, CGLM06, DM09], specifying both the permitted behaviour and the behaviour of the algorithm using I/O automata (IOAs) [LT87], and using

simulation proof techniques [LV95] to show that an algorithm implements a given specification. This approach supports hierarchical reasoning, because an automaton can describe both an algorithm implementing some higher-level specification and a specification for a more detailed implementation. By constructing these proofs using the PVS verification system [PVS], our proofs can be entirely machine-checked, greatly reducing the likelihood of errors in our proofs. For example, when a reviewer pointed out that one property we had proved was unnecessary, we were able to confirm this by removing the property and rerunning the remaining proof; such a change in a manual proof would carry a greater risk of introducing an error. We aim to share PVS models and proof scripts, and to encourage others to similarly contribute. By enabling the sharing and reuse of proofs, we hope to make it significantly easier to formally establish the correctness of different TM implementations and variants, without repeating all of the hard work of a bottom-to-top proof.

In defining a correctness condition for TM, we have several goals beyond providing an unambiguous and precise definition that supports formal machine-checkable proofs: The condition must make sufficient semantic guarantees to be useful to programmers using TM, but should be as permissive as possible, consistent with these guarantees, to avoid arbitrarily excluding implementation techniques, including ones not yet invented. Finally, it should be easy to understand and reason about, so that researchers and implementors can prove that TM implementations satisfy the condition.

There is tension between these goals: the generality of a highly permissive correctness condition makes the condition harder to understand, while admitting additional behaviours that are exhibited by few, if any, real implementations. We deal with this tension by specifying multiple correctness conditions. Specifically, in addition to TMS1, we present TMS2 (see Section 5), which is less permissive, and closer to the intuition of how many of the existing practical TM runtime libraries work. We also prove that TMS2 implements TMS1 (see Section 6), and we have formalised and verified this proof using PVS. This reduces the task of proving that a TM implementation satisfies the TMS1 condition to the easier task of proving that it satisfies TMS2.

This "hierarchical" proof style can be extended and refined further. For example, in ongoing work, we are proving that some existing TM implementations are correct in stages, as follows: First, we consider a simplified model of a TM implementation that uses coarser grained synchronisation than is available in existing systems, and prove that it implements TMS2. Next, we prove that a more realistic model implements this simplified one. Putting together all of these proofs, we can formally establish that a realistic model of a practical TM implementation correctly implements the semantics specified by TMS1. We discuss our ongoing and intended future work in more detail in Section 8.

2. Preliminaries

This section provides background on how we express TM correctness conditions, how we model TM implementations, and how we prove relationships between them.

2.1. Serialisations and partial functions

We denote the set of finite sequences of a set S by S^*, and the concatenation of finite sequences σ_1 and σ_2 by $\sigma_1 \cdot \sigma_2$.

Given a finite set S, a *serialisation* of S is a sequence $\sigma \in S^*$ such that each element of S occurs exactly once in σ. We denote the set of serialisations of S by $ser(S)$. For a serialisation σ of S and any element $s \in S$, we denote the prefix of σ up to and including s by $\sigma|_{\leq s}$.

A serialisation σ of S implicitly defines a reflexive total order $\leq_\sigma$ on S, where $(a, b) \in \leq_\sigma$ if a is in $\sigma|_{\leq b}$. A serialisation of S *respects* a relation R if the restriction of R to S is contained in this total order (i.e., if $R \cap (S \times S) \subseteq \leq_\sigma$). We denote the set of such serialisations by $ser(S, R)$. (If R forms any cycle in S, then $ser(S, R)$ is empty.)

We represent a partial function f from X to Y by a function from X to $Y_\perp = Y \cup \{\perp\}$ that maps elements on which f is not defined to $\perp$. We abuse notation by writing $f : X \to Y_\perp$. We denote the domain of f by $\mathbf{dom}(f) = \{x \in X \mid f(x) \neq \perp\}$.

Given a (partial or total) function f from X to Y, and $x \in X$ and $y \in Y$, we denote by $f[x \to y]$ the function f' such that $f'(x) = y$ and $f'(x') = f(x')$ for all $x' \neq x$. For $g : X \to Y_\perp$, we denote by $f[g]$ the function $f' : X \to Y$ such that $f'(x) = g(x)$ for $x \in \mathbf{dom}(g)$ and $f'(x) = f(x)$ for $x \notin \mathbf{dom}(g)$.

2.2. I/O automata

We use *input/output automata* (IOAs) [LT87] to express TM correctness conditions and to model TM implementations. An IOA A is a labeled transition system that consists of: a set $states(A)$ of states; a nonempty set $start(A) \subseteq states(A)$ of start states; a set $acts(A)$ of actions; a signature $sig(A) = (in(A), out(A), internal(A))$, which partitions $acts(A)$ into *input*, *output*, and *internal* actions; and a transition relation $trans(A) \subseteq states(A) \times acts(A) \times states(A)$. We call the actions in $in(A) \cup out(A)$ *external actions*. (For this paper, we do not include a fairness partition because we are concerned only with safety.) An IOA must be *input-enabled*: for every state $s \in states(A)$ and every input action $a \in in(A)$, there must be some state $s' \in states(A)$ such that $(s, a, s') \in trans(A)$. We typically describe the states using a collection of *variables*, and the transition relation using a *precondition* (a predicate on states) and an *effect* (a set of assignments to variables) for each action.

An *execution fragment* of A is a sequence $s_0 a_1 s_1 \ldots$ of alternating states and actions of A, such that $(s_{k-1}, a_k, s_k) \in trans(A)$ for all k; a finite execution fragment must end with a state. An *execution* is an execution fragment with $s_0 \in start(A)$. A state is *reachable* if it appears in some execution. An *invariant* is a predicate that is true for all reachable states; it is typically proved by induction on the length of an execution.

The subsequence of external actions in an execution fragment is called its *trace*, and represents its externally visible *behaviour*. An instance of an action in a trace is called an *event*. The traces of an automaton A are the traces of its executions; we denote the set of such traces by $traces(A)$. For an "abstract" automaton A, modelling a specification, and a "concrete" automaton C, modelling an implementation, C *implements* A iff $traces(C) \subseteq traces(A)$: every behaviour of the implementation is allowed by the specification.

2.3. Simulation proofs

One way to prove that C implements A is via a *forward simulation* from C to A [LV95], which is a relation R between $states(C)$ and $states(A)$ such that:

- for every $s \in start(C)$ there is a $u \in start(A)$ such that $(s, u) \in R$, and
- for every reachable state $s \in states(C)$, every reachable state $u \in states(A)$ such that $(s, u) \in R$, and every step $(s, a, s') \in trans(C)$, there is an execution fragment π of A starting from u and ending in a state u' such that
  - $(s', u') \in R$, and
  - $trace(\pi) = trace(a)$ (that is, if a is an internal action, then the execution fragment π has only internal actions; if a is an external action, then π contains that action and no other external actions).

A forward simulation that is a function on the reachable states of C is called a *refinement* [LV95].

A forward simulation from C to A proves that any trace of C is also a trace of A: Given an execution of C, we can construct an execution of A with the same trace by choosing a start state of A related to the start state of C's execution by the forward simulation, and then for every step of C in turn, extending the execution of A with the execution fragment required by the forward simulation.

Lemma 1 If there exists a forward simulation from C to A then C implements A.

2.4. Objects and their sequential semantics

To state and prove properties of TM implementations that support general objects, we first define *object types*, which specify the *interface* and *sequential semantics* of an object. Given multiple such object types, it is straightforward to define a single object type supporting all of their operations with the same semantics. Therefore, without loss of generality, we henceforth consider TM systems that support a single object type.

The interface of an object type $\mathcal{O}$ consists of a set $\mathcal{I}_\mathcal{O}$ of possible *operation invocations* and a set $\mathcal{R}_\mathcal{O}$ of possible *operation responses*. When there is no ambiguity, we sometimes just use *invocations* and *responses*. An *operation* of the object is an invocation-response pair. The *sequential semantics* of an object type $\mathcal{O}$ specifies which sequences of operations (i.e., elements of $(\mathcal{I}_\mathcal{O} \times \mathcal{R}_\mathcal{O})^*$) are *legal sequential histories*; the predicate $legal_\mathcal{O}$ distinguishes such sequences.

2.5. Transactions

To model transactions, we use a set $\mathcal{T}$ of transaction identifiers. We model a *transactional memory system that supports object type* $\mathcal{O}$ as an automaton $TM_{\mathcal{O}}$ with:

$$in(TM_{\mathcal{O}}) = \{\mathsf{begin}_t \mid t \in \mathcal{T}\} \cup \{\mathsf{inv}_t(i) \mid i \in \mathcal{I}_{\mathcal{O}}, t \in \mathcal{T}\} \cup \{\mathsf{commit}_t \mid t \in \mathcal{T}\} \cup \{\mathsf{cancel}_t \mid t \in \mathcal{T}\}$$
$$out(TM_{\mathcal{O}}) = \{\mathsf{beginOk}_t \mid t \in \mathcal{T}\} \cup \{\mathsf{resp}_t(r) \mid r \in \mathcal{R}_{\mathcal{O}}, t \in \mathcal{T}\} \cup \{\mathsf{commitOk}_t \mid t \in \mathcal{T}\} \cup \{\mathsf{abort}_t \mid t \in \mathcal{T}\}$$

We refer to the actions in $in(TM_{\mathcal{O}})$ as *invocations*, and those in $out(TM_{\mathcal{O}})$ as *responses*. We require that every transactional memory system satisfies the following basic well-formedness property: Responses occur only after a corresponding invocation for which there has not yet been a matching response. The system may, however, abort a transaction in response to any invocation of that transaction. More precisely, in any well-formed execution and for each transaction $t \in \mathcal{T}$:

- A $\mathsf{beginOk}_t$ (respectively $\mathsf{resp}_t(r)$ or $\mathsf{commitOk}_t$) occurs only if there is a preceding begin_t (respectively $\mathsf{inv}_t(i)$ or commit_t) with no subsequent response of t.
- An abort_t occurs only if there is a preceding invocation of t with no subsequent response of t.

A transactional memory implementation typically imposes well-formedness conditions on its clients, which we can model using a *clients automaton* whose output actions are the invocations and whose input actions are the responses of the TM. To model the entire system, the clients automaton must be composed with a transactional memory automaton. In this paper, we require that begin_t is the first invocation of transaction t, that transactions wait to receive a response to each invocation before its next invocation, and that no committed or aborted transaction invokes an operation. More precisely, we have the following well-formedness conditions on the clients in any execution and for each transaction $t \in \mathcal{T}$:

- No invocation of t occurs before begin_t. (This implies that at most one begin_t occurs.)
- After an invocation by t, no invocation of t occurs until a response of t occurs.
- No invocation of t occurs after $\mathsf{commitOk}_t$ or abort_t.

We say that a transaction is *active* if it has invoked begin, has invoked neither commit nor cancel, and has not received an abort response.

For simplicity, when we define a TM system (or correctness condition), we present a single automaton that is isomorphic to the composition of the automaton for that system with the clients automaton. (See [LT89] for formal details of how IOAs are composed.) Figure 1 presents such an automaton that guarantees *only* the basic well-formedness property described above. The well-formedness conditions on the clients are expressed in the preconditions of the invocations, and the well-formedness of the system is guaranteed by the preconditions on the responses (the combined automaton has no input actions because inputs to the transactional memory are outputs of the clients; therefore, such automata are trivially input-enabled).

2.6. Serial executions

As discussed in Section 1, transactions allow programmers to reason about their programs as if transactions are executed one at a time, even though the underlying implementation generally does allow transactions to execute concurrently for performance reasons. To make this requirement more precise, we define a *serial execution* as an execution in which all events associated with each transaction are consecutive, and each transaction, except possibly the last one, has either committed or aborted.

In Section 3.2, we explain why our TMS1 correctness condition guarantees that a transactional program can never detect that transactions are not executed serially. This explanation describes how, given an execution of a transactional program, we can use the properties of TMS1 to construct a serial execution of that program (i.e., one in which the steps of a transaction are not interleaved with any other steps in the execution) such that the program cannot distinguish between the serial execution and the actual execution.

To keep the automata that we use to specify TM correctness conditions as simple as possible, they do not explicitly model all of the non-TM-related details of a transactional program execution, such as local computation, updating of registers, and nontransactional memory accesses, synchronisation, and communication. It is not necessary to model these details in order to specify our TM correctness conditions, at least in our context, due to the following assumptions and observations:

State variables

For each $t \in \mathcal{T}$:
$status_t$: {notStarted, beginPending, ready, opPending, commitPending, cancelPending, committed, aborted}; initially notStarted

Actions for each $t \in \mathcal{T}$

begin_t	$\mathsf{beginOk}_t$
Pre: $status_t = \mathsf{notStarted}$	Pre: $status_t = \mathsf{beginPending}$
Eff: $status_t \leftarrow \mathsf{beginPending}$	Eff: $status_t \leftarrow \mathsf{ready}$
$\mathsf{inv}_t(i),\ i \in \mathcal{I}_{\mathcal{O}}$	$\mathsf{resp}_t(r),\ r \in \mathcal{R}_{\mathcal{O}}$
Pre: $status_t = \mathsf{ready}$	Pre: $status_t = \mathsf{opPending}$
Eff: $status_t \leftarrow \mathsf{opPending}$	Eff: $status_t \leftarrow \mathsf{ready}$
commit_t	$\mathsf{commitOk}_t$
Pre: $status_t = \mathsf{ready}$	Pre: $status_t = \mathsf{commitPending}$
Eff: $status_t \leftarrow \mathsf{commitPending}$	Eff: $status_t \leftarrow \mathsf{committed}$
cancel_t	abort_t
Pre: $status_t = \mathsf{ready}$	Pre: $status_t \in \{\mathsf{beginPending}, \mathsf{opPending}, \mathsf{commitPending}, \mathsf{cancelPending}\}$
Eff: $status_t \leftarrow \mathsf{cancelPending}$	Eff: $status_t \leftarrow \mathsf{aborted}$

Fig. 1. An automaton capturing the basic well-formedness guarantees and requirements for a TM system that supports object type $\mathcal{O}$. All actions are external.

First, we assume that when a transaction aborts, client code (perhaps compiler-generated) takes care of undoing changes to local variables and registers caused by computation within the transaction.

Second, as discussed in Section 1, we assume for this paper that correct transactional programs do not allow transactions to access nontransactional variables, or nontransactional code to access transactional variables, and that transactions do not otherwise communicate with the outside world. We also assume that nontransactional code does not access internal state of the TM implementation, and in particular, it cannot determine whether a transaction has begun.

Together these assumptions imply that information about the behaviour of an aborted transaction can become known outside the transaction only via the TM interface. That is, no information can "leak" out of an aborted transaction.

Furthermore, these assumptions imply that local steps within a transaction (i.e., computation, updating registers, etc.) depend only on the contents of local variables and registers at the beginning of the transaction (which represent any "knowledge" the transaction has about past execution) and interactions with the TM implementation via its interface. Therefore, if we can show, for a given transaction, that the contents of local variables and registers and the interactions with the TM are the same in the constructed serial execution as in the actual execution, then the local actions behave the same in both executions, so there is no need to model them explicitly.

We can similarly avoid the need to explicitly model nontransactional events (i.e., events that do not occur within a transaction) in the actual execution. In particular, if we ensure that the real-time order of nontransactional events relative to each other and to transactions in the actual execution is preserved in the constructed serial execution, and that all previously completed transactions receive the same responses in both executions, then again we can conclude that such actions behave the same way in both executions, and therefore they do not need to be modeled explicitly in our automata. (In the *real-time order* for a given execution, also known as the *external order*, a transaction t_1 *precedes* a transaction t_2 iff t_1's final event (an abort or successful commit) comes before t_2's first event (begin). The real-time order orders two nontransactional events in the same order they appear in the execution. A nontransactional event precedes a transaction in the real-time order iff it occurs before the transaction begins, and it follows the transaction iff it occurs after the transaction has committed or aborted in the actual execution. Note that events that occur during the transaction's interval are not ordered with respect to the transaction, and may therefore occur either before or after the transaction in the constructed serial execution.) Critical to this argument is the requirement that the order of transactions in the constructed serial execution respects the real-time order of the transactions in the actual execution. Otherwise, we might not be able to preserve the order of all nontransactional events relative to transactions in the serial execution.

2.7. Read-write memory

Most TM implementations assume a specific type of object called *read-write memory*. For this reason, our TMS2 automaton is specialised for read-write memory, which we now define formally. Read-write memory (RW) is an object type parameterised by a nonempty set L of *locations*, a nonempty set V of *values*, and a nonempty *initialisation predicate* *init* over *memory states*, which are functions from L to V. The interface of a read-write memory object is

$$\mathcal{I}_{\mathrm{RW}(L,V,init)} = \{read(l) \mid l \in L\} \cup \{write(l,v) \mid l \in L, v \in V\}$$
$$\mathcal{R}_{\mathrm{RW}(L,V,init)} = V \cup \{\mathrm{ok}\}$$

For the rest of the paper, we fix L and V, and specify only the initialisation predicate. A sequence of operations is a legal sequential history of an RW(*init*) object if and only if there is a memory state m satisfying the initialisation predicate *init* such that the response of every write operation is "ok" and the response of a read operation $read(l)$ is the value written by the last write to the same location preceding that read operation in the sequence, or $m(l)$ if there is no such write.

It is often convenient to think of legal sequential histories of RW(*init*) in terms of sequences of memory states between operations, as formalised by the following lemma.

Lemma 2 A finite sequence $ops = op_1 op_2 \dots op_n$ of operations of RW(*init*) is a legal sequential history of RW(*init*) if and only if there is a sequence $m_0 m_1 m_2 \dots m_n$ of memory states such that $init(m_0)$ and for all k such that $0 < k \leq n$,

- if $op_k = (write(l,v), r)$ then $r = \mathrm{ok}$ and $m_k = m_{k-1}[l \to v]$, and
- if $op_k = (read(l), r)$ then $r = m_{k-1}(l)$ and $m_k = m_{k-1}$.

The *result of applying an operation* to a memory state m is m if the operation is a $read(l)$ operation for any l, and $m[l \to v]$ if the operation is a $write(l,v)$ operation. Given a memory state m of RW(*init*), and a finite sequence $ops = op_1 \dots op_k$ of operations, we say that ops is *legal starting from* m if it is a legal sequential history of RW($init_m$), where $init_m(m') \iff m = m'$, and that the *result of applying* ops to m is the memory state that results from applying op_k to the result of applying $op_1 \dots op_{k-1}$ to m (if ops is empty then the result of applying ops to m is just m). We say that *applying ops to m legally results in* m', and write $m \xrightarrow{ops} m'$, if ops is legal starting from m and the result of applying ops to m is m'. The following lemmas capture straightforward properties related to these definitions that are needed in our proof.

Lemma 3 If ops is legal starting from m and $init(m)$ then ops is a legal sequential history of RW(*init*).

Lemma 4 If m, m' and m'' are memory states and ops_1 and ops_2 are finite sequences of operations of a read-write memory (with any initialisation predicate) such that $m \xrightarrow{ops_1} m'$ and $m' \xrightarrow{ops_2} m''$, then $m \xrightarrow{ops_1 \cdot ops_2} m''$.

Lemma 5 If m and m' are memory states and ops_1 and ops_2 are finite sequences of operations of a read-write memory (with any initialisation predicate) such that $m \xrightarrow{ops_1 \cdot ops_2} m'$, then there exists a memory state m'' such that $m \xrightarrow{ops_1} m''$ and $m'' \xrightarrow{ops_2} m'$.

Given a finite sequence of operations ops, we define the partial function $writes(ops) : L \to V_\perp$ to map each location written in ops to the last value written to it in ops if there is one, and to $\perp$ otherwise.

Lemma 6 For a memory state m and a finite sequence ops of operations of a read-write memory (with any initialisation predicate), the result of applying ops to m is $m[writes(ops)]$ (i.e., the function that maps each location l to $writes(ops)(l)$ if $l \in \mathbf{dom}(writes(ops))$, and to $m(l)$ otherwise).

3. TMS1: a correctness condition for TM

In this section, we present the TMS1 correctness condition for TM; we relate it to previous conditions in the next section. TMS1 is based on the following key design principles:

- No transaction observes partial effects of any other transaction.
- At any point in time, there must be (at least) one serialisation that includes all committed transactions and no aborted transactions, such that concatenating the operations of the transactions in the order of this serialisation yields a legal sequential history. We say that such a serialisation "justifies" the transactions' responses.
- To ensure that a transaction observes behaviour that is consistent with being part of *some* serial execution even if the transaction subsequently aborts, responses to operations executed by an active transaction before it invokes commit must also be justified by a serialisation.
- To avoid unnecessarily precluding implementations, we want to constrain the choices of serialisations used to justify transactions' responses as little as possible. Nonetheless, some constraints are necessary to ensure that a transaction cannot observe behaviour that *could not have* happened in any serial execution. We discuss these constraints below.

For the purposes of the first principle above, the effects of a transaction are regarded as partial until it invokes commit, at which point we say the transaction becomes *visible*. This is because we assume that all communication between a transactional program and a TM runtime is via the specified interface, so there is no way for the TM runtime to know what a transaction will or will not do in the future. We discuss ways in which our interface and condition could be modified to take advantage of additional knowledge, for example from compiler analysis, in Section 8.

The second principle ensures that it is possible to interpret the observed behaviour of a transactional program as the result of the program executing without interleaving operations of any transactions (i.e., as the behaviour exhibited by some serial execution), so that a transactional programmer need only consider such executions when reasoning about the program.

The third principle is motivated by the recognition that, in an unmanaged environment, it is not sufficient to ensure that a transaction t does not commit successfully if its responses cannot be justified by the serialisation that justifies the responses of other transactions. If t observes a pattern of responses that *no* serial execution could justify, it might execute an operation that causes an unrecoverable error (such as divide-by-zero) before t even attempts to commit. By requiring the responses that t observes to be consistent with the current execution *from t's point of view*, we ensure that such an error occurs only if there really is a bug in the program, such that the error could happen in *some* serial execution (i.e., the execution implied by the serialisation that justified the behaviour of the errant transaction).

Note that the responses of different transactions need not be justified by the same serialisation; each active or aborted transaction may have its responses justified by a different serialisation, and these serialisations need not be consistent with each other or with the one used to justify the responses of the committed transactions. This is because, while a transaction is active, no other transaction can infer what serialisation was used to justify its responses before it completed. Furthermore, if it aborts, no other transaction can ever infer that serialisation.

In accordance with the fourth principle, we want to place as few restrictions as possible on the serialisations used to justify the responses of a transaction because we do not want to unnecessarily restrict implementations. For example, an implementation may have an optimistic commit protocol that makes effects of a committing transaction visible before the transaction is guaranteed to commit. Such an implementation may still be correct if it is *dependence-aware* [AA08, RRW08], so that if such a transaction ultimately aborts, any transaction that sees its effects must also abort (unless its responses can be justified by some serialisation that does not include the aborted transaction). Thus, for example, the execution illustrated in Figure 2 should be allowed.

Furthermore, we do not require the serialisation used to justify the responses of an *active* transaction to include *all* committed transactions, and it may even include some aborted transactions, provided that those transactions are visible (i.e., have invoked commit) in accordance with the first principle above. Nor do we require the serialisation that justifies the responses of a transaction to be fixed: as long as at any time, *some* serialisation justifies the transaction's responses, there is no reason the serialisation that does so cannot change over time. This point is illustrated in Figure 3.

As we shall see, this flexibility implies that a nontrivial condition must be satisfied in order to *abort* a transaction. This may be somewhat surprising because, while all correctness conditions impose constraints on the conditions under which a transaction can commit, most allow any attempt to commit a transaction to fail. However, this simple property precludes implementations in which one transaction can commit having observed the effects of another that has not yet committed (as in the above example).

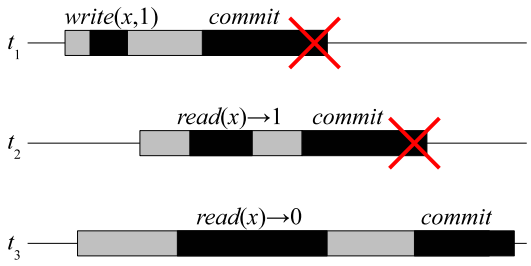

Fig. 2. An execution illustrating that a transaction may see the effects of another transaction that ultimately aborts. The actions of each transaction are illustrated on a line, where time increases to the right. Grey boxes represent transactions; black boxes represent operations within those transactions (we omit the begin operations, which can be inferred from where the transaction box begins). Return values are shown only for *read* operations. A large X marks when a transaction aborts. In this and all other example executions in this paper, we assume a read-write memory whose locations all initially have value 0.

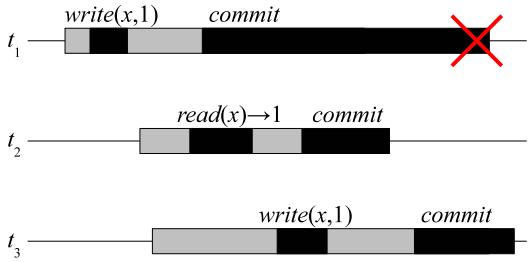

Fig. 3. An execution illustrating that the serial execution used to justify a response may change over time: When t_2's read returns, its response can be justified only by t_1's write. However, even though t_2 commits, t_1 can abort after t_3 makes its write visible by invoking commit. (Until that point, t_1 cannot abort because if it did, t_2's behaviour could not be justified.)

Despite this flexibility, as mentioned above, we must constrain the serialisations used to justify the responses of transactions in order to prevent transactions from detecting inconsistencies, and causing unrecoverable errors as a result. For example, a thread might execute a transaction only after it commits a prior transaction. In that case, any serialisation used to justify the responses of the later transaction must include the earlier one. Similarly, if the earlier transaction aborts, then a serialisation used to justify the responses of the later transaction must *not* include the earlier one.

Moreover, in practical systems for which we intend TMS1, we expect that transactions will not necessarily be the only means of coordination. For example, whenever one transaction t completes (successfully or not) before another transaction t' begins, the threads executing the transactions may communicate between the end of t and the beginning of t', so that they know that t was executed before t'. Indeed, t' might be executed *in response to* the result of the execution of t. Because the TM runtime has no knowledge of such communication or how it may affect the program's behaviour, a serialisation used to justify the responses of these and other transactions must order t before t' to avoid the program detecting an inconsistency. Thus, we require these justifying serialisations to respect the external order of transactions; we call this property *external consistency*. Ordering transactions executed by the same thread is a special case of this requirement; therefore no explicit notion of threads is needed.

3.1. Formal definition of TMS1

We now present a formal definition of the TMS1 correctness condition, designed in accordance with the principles above. In particular, we define TMS1 as the set of traces produced by the automaton shown in Figure 4. Expressing our conditions this way serves both to make them unambiguous, and to facilitate the construction of precise, machine-checked proofs about them.

The TMS1 automaton is mostly self-explanatory. Each action updates one or more bookkeeping variables

State variables

$extOrder$: binary relation on $\mathcal{T}$; initially empty
For each $t \in \mathcal{T}$:
  $status_t$: {notStarted, beginPending, ready, opPending, commitPending, cancelPending, committed, aborted}; initially notStarted
  ops_t: $(\mathcal{I}_\mathcal{O} \times \mathcal{R}_\mathcal{O})^*$ (i.e., a sequence of operations); initially empty
  $pendingOp_t$: $\mathcal{I}_\mathcal{O}$; initially arbitrary
  $invokedCommit_t$: Boolean; initially **false**

Actions for each $t \in \mathcal{T}$

begin_t
Pre: $status_t = \mathsf{notStarted}$
Eff: $status_t \leftarrow \mathsf{beginPending}$
  $extOrder \leftarrow extOrder \cup (DT \times \{t\})$

$\mathsf{beginOk}_t$
Pre: $status_t = \mathsf{beginPending}$
Eff: $status_t \leftarrow \mathsf{ready}$

$\mathsf{inv}_t(i)$, $i \in \mathcal{I}_\mathcal{O}$
Pre: $status_t = \mathsf{ready}$
Eff: $status_t \leftarrow \mathsf{opPending}$
  $pendingOp_t \leftarrow i$

$\mathsf{resp}_t(r)$, $r \in \mathcal{R}_\mathcal{O}$
Pre: $status_t = \mathsf{opPending}$
  $validResp(t, pendingOp_t, r)$
Eff: $status_t \leftarrow \mathsf{ready}$
  $ops_t \leftarrow ops_t \cdot (pendingOp_t, r)$

commit_t
Pre: $status_t = \mathsf{ready}$
Eff: $status_t \leftarrow \mathsf{commitPending}$
  $invokedCommit_t \leftarrow \mathbf{true}$

$\mathsf{commitOk}_t$
Pre: $status_t = \mathsf{commitPending}$
  $validCommit(t)$
Eff: $status_t \leftarrow \mathsf{committed}$

cancel_t
Pre: $status_t = \mathsf{ready}$
Eff: $status_t \leftarrow \mathsf{cancelPending}$

abort_t
Pre: $status_t \in \{\mathsf{beginPending}, \mathsf{opPending}, \mathsf{commitPending}, \mathsf{cancelPending}\}$
  $validFail(t)$
Eff: $status_t \leftarrow \mathsf{aborted}$

Derived state variables, functions and predicates

$$DT \triangleq \{t \mid status_t \in \{\mathsf{committed}, \mathsf{aborted}\}\}$$
$$CT \triangleq \{t \mid status_t = \mathsf{committed}\}$$
$$CPT \triangleq \{t \mid status_t = \mathsf{commitPending}\}$$
$$VT \triangleq \{t \mid invokedCommit_t\}$$
$$ops(\sigma) \triangleq ops_{t_1} \cdot ops_{t_2} \cdot \ldots \cdot ops_{t_n} \text{ where } \sigma = t_1 t_2 \ldots t_n$$
$$extConsPrefix(S) \triangleq \forall t, t' \in \mathcal{T},\ t' \in S \wedge (t, t') \in extOrder \implies (t \in S \iff status_t = \mathsf{committed})$$
$$validCommit(t) \triangleq \exists S \subseteq CPT, \exists \sigma \in ser(CT \cup S, extOrder),\ t \in S \wedge legal_\mathcal{O}(ops(\sigma))$$
$$validFail(t) \triangleq \exists S \subseteq CPT, \exists \sigma \in ser(CT \cup S, extOrder),\ t \notin S \wedge legal_\mathcal{O}(ops(\sigma))$$
$$validResp(t, i, r) \triangleq \exists S \subseteq VT, \exists \sigma \in ser(S, extOrder),\ extConsPrefix(S \cup \{t\}) \wedge legal_\mathcal{O}(ops(\sigma \cdot t) \cdot (i, r))$$

Fig. 4. The $\mathsf{TMS1}(\mathcal{O})$ automaton, a TM that supports object type $\mathcal{O}$. All actions are external.

in straightforward ways. The *extOrder* variable encodes the external order of transactions: each time a new transaction begins, the begin action records that it occurs after every transaction that has already completed (i.e., committed or aborted).

The interesting aspects of this automaton are in three "validation" conditions imposed on the possible responses of the system. (A new transaction may always begin, so beginOk has no corresponding validation condition.) Specifically, for a TM system to return a response to either an operation or an attempt to commit (successful or otherwise), it must ensure that this response can be justified by some serialisation. As discussed above, different constraints apply to these serialisations for different cases; the constraints are encoded in the validation conditions for the response actions.

The validation conditions for all responses (except beginOk) require the existence of a serialisation of a set of visible transactions that respects the external order of transactions, such that concatenating the operations in the transactions in the order of the serialisation yields a legal sequential history. The differences

between the different conditions lie in what transactions can and cannot be included in these serialisations, as discussed above.

The validation condition (*validCommit*) for committing a transaction t requires that the serialisation used to justify a $\mathsf{commitOk}_t$ response include t and all transactions that have already committed (CT). It may also include some transactions that have invoked commit but not yet committed or aborted; we call these transactions *commit-pending*.

The validation condition (*validFail*) for aborting transaction t is identical to that for committing t, except that the serialisation that justifies the response must *not* include t: for t to abort safely, it must be possible to justify the successful commit of all transactions that have already committed *without* using t.

The validation condition (*validResp*) for returning a response to an operation within a transaction t is trickier. As discussed above, we have considerable freedom in choosing the serialisation used to justify a response to an operation, but nonetheless some constraints are needed to ensure that the transaction does not detect an inconsistency. In particular, the set of transactions included in the serialisation used to justify a response of an active transaction t must be an externally consistent prefix set consisting of t and some subset of the visible transactions. (Given a set of transactions and an external order over these transactions, we say that a subset S of transactions is an *externally consistent prefix set* if, for any transactions t and t' such that t is externally ordered before t', if t' is in S then t is in S if and only if t is committed. This requirement is formalised by the *extConsPrefix* predicate. The intuition behind this definition is explained in the next section.)

3.2. Why TMS1 enables transactional programming

The purpose of TMS1 is to specify what guarantees the TM runtime must make in order to ensure that programmers who think about their programs as if only serial executions (i.e., executions in which the events of each transaction appear consecutively) are possible do not receive any unpleasant surprises as a result of the concurrent execution of transactions. We explain below how TMS1's validation conditions ensure that all responses given by the TM runtime are consistent with some serial execution of the program. In particular, for each response, we describe how to transform the actual program execution into a serial execution (i.e., one in which transactions are not interleaved with each other) such that the program cannot distinguish between the actual execution and the constructed serial execution.

First consider a commitOk or abort response that occurs when there are no other commit-pending transactions. The validation conditions for these actions require the response to be justified by a serialisation of exactly the set of transactions that have committed successfully so far (including the one receiving the response if it is a commitOk response) such that the serialisation respects the external order of transactions, and that concatenating the operations performed within these transactions in the order of this serialisation yields a legal sequential history.

The transformation to a serial execution in this case is straightforward: First we remove all events of each transaction that is active when the response is received. Doing so does not otherwise affect the execution, due to the assumptions that active transactions do not communicate with the outside world, the outside world does not communicate with active transactions (see Section 2.6), and the fact that active transactions never appear in the serialisation used to justify any TM response (see Figure 4). For similar reasons, for each aborted transaction (other than the one receiving the response if it is an abort response), we can remove all its events except its begin and abort actions, without affecting the rest of the execution. Because the serialisation used to justify the response under consideration is required by the relevant validation condition (*validCommit* or *validFail*) to respect the external order, we can now rearrange the execution into one in which the events of each committed transaction occur consecutively (that is, they are not interleaved with other events) at some point during its execution, while otherwise leaving the program execution undisturbed. (This reordering is possible because a transaction is not allowed to access shared memory or use any means of communicating externally other than via TM invocations and responses; see Section 2.6.)

After the rearrangement described above, all nontransactional actions are still ordered the same with respect to each other as they were in the original execution (because we did not change their order). Importantly, the order of these actions with respect to *transactions* can also be preserved by the transformation, due to the requirement that the serialisation respect the external order. Thus, as discussed previously, these actions behave exactly the same in the constructed serial execution as they do in the actual execution.

Thus, for this restricted case, it is easy to construct a serial execution and to see that the program cannot distinguish the actual execution from the serial execution.

Now consider a commitOk or abort response that occurs when there *are* other commit-pending transactions. In this case, the validation conditions allow a subset of these commit-pending transactions to be included in the serialisation that justifies the response, even though they may subsequently abort. However, because such transactions are commit-pending, they *could* have committed already (note that validation conditions require that these transactions are included in the serialisation in a way that respects the external order of transactions and the sequential semantics). Furthermore, because such transactions are commit-pending, the program does not (yet) know their outcome, and therefore cannot have done anything to allow the response under consideration to detect a difference between the actual execution and one in which the transaction has committed. (If such a commit-pending transaction later aborts, the validation condition for that abort guarantees that the response we are considering is still justified by the serialisation used to justify the later abort.) Similarly, each commit-pending transaction *not* included in the justifying serialisation could have been aborted already. Thus, we can transform the execution by adding a commitOk response for each commit-pending transaction included in the justifying serialisation and an abort response for each one not included (each of these responses is justified by the same serialisation used to justify the response under consideration), and then applying the transformation described for the previous case.

Finally, consider a response to an operation invocation within an active transaction t^*. Because t^* might subsequently be aborted and we assume that no information leaks out of an aborted transaction other than the fact that it aborted (see Section 2.6), the behaviour t^* observes need not be consistent with the serialisation that justifies the responses of the committed transactions. Nonetheless, as already discussed, it is critical in our target context that the behaviour observed by t^* could have occurred in *some* serial execution. Otherwise, t^* may cause an unrecoverable error, such as divide-by-zero, before it discovers that it cannot commit.

As in the cases of commitOk and abort responses, given an execution of a transactional program in which t^* receives a new operation response, and a justifying serialisation that satisfies the *validResp* condition, we can transform the execution into a serial execution of the program that has the same behaviour from the point of view of t^*. However, the transformation is more involved, as is the argument that the resulting serial execution is consistent with the behaviour observed by t^*, because the validation condition for an operation response is more permissive than for commitOk and abort responses.

In particular, the *validResp* condition allows the justifying serialisation to include some transactions that have *already* aborted, and similarly to exclude some that have already committed. The transformation must change the responses of any such transactions in the execution. If no response needs to be changed, then we can transform the execution in much the same way as in the previous case: remove any events of active transactions other than t^*, add a commitOk or abort response for each commit-pending transaction, depending on whether the transaction appears in the justifying serialisation, and then rearrange the remaining events into a serial execution (with t^* as the final, and only active, transaction).

If the response of a transaction t must be changed, we must ensure that this change is not observable to any transaction (other than t^*, for which the change is required) or nontransactional event that remains in the transformed serial execution. We can do so by removing any event that might detect this change, that is, any event that is externally ordered after the altered response. But what if the justifying serialisation includes a transaction t' that begins after t completes? In that case, t^*'s response may be one that could not occur in *any* serial execution of the program, because it may be possible to infer from the response that the execution of t' is not consistent with the outcome for t used to justify the response. It is for exactly this reason that we cannot allow the serialisation to differ from the actual execution on the outcome of transaction t if we include another transaction (such as t') that follows it. Specifically, *extConsPrefix* requires that, if a transaction t' is included in the serialisation, then we cannot treat any transaction that precedes it differently from the actual execution: every such transaction must be included in the serialisation if it has committed in the actual execution, and excluded if it has aborted.

4. Comparing TMS1 to previous closely related correctness conditions

We presented a preliminary version of TMS1, which we called WRC, for the "weakest reasonable condition", at the 2009 Refinement Workshop [DGLM09]. We subsequently realised that we could weaken the condition further without violating the principles given in the previous section. Furthermore, we recognise that what

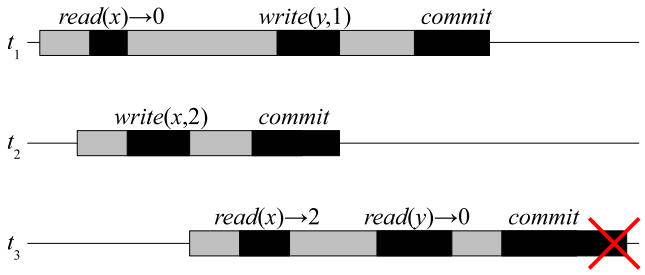

Fig. 5. An execution illustrating how opacity unnecessarily constrains the order of transactions used to justify the behaviour observed by transactions that abort: This execution is not opaque because the only serialisation allowed for committed transactions is $t_1 t_2$, while t_3 observes that t_2 has committed but t_1 has not. TMS1 and VWC allow this execution because the responses t_3 sees are consistent with behaviour that *could* have occurred, namely that t_1 could have aborted.

is reasonable is context-dependent, and thus have resorted to using generic names rather than try to find short names with meanings that are valid in limited contexts.

We defined TMS1 in part to overcome shortcomings of *opacity* [GK08, GK10]. Opacity was the first attempt to formally define a correctness condition that requires all transactions (even transactions that abort) to observe consistent behaviour, as TMS1 does. The model in which opacity is expressed has a similar interface and similar assumptions as we have in this paper, except that we have an explicit begin_t action for transactions, a mostly cosmetic difference. We believe (but have not proved) that opacity implies TMS1.

The most substantive difference between TMS1 and opacity is that TMS1 allows different aborted transactions to disagree on which other transactions commit, and in what order. Thus, opacity precludes implementations that a programmer cannot distinguish, from one that satisfies opacity, without violating rules (often enforced by the compiler) prohibiting leaking information out of aborted transactions (see Section 2). Precluding such implementations violates the fourth design principle in Section 3.

One other interesting subtlety is that, in its original definition [GK08], opacity is not prefix-closed, and therefore is not a safety property. For example, a read executed by one transaction might read a value written later by a concurrent transaction. In this case, even though the entire execution, in which both transactions have committed, may satisfy opacity, the prefix of the execution up to the read does not. To address this problem, Guerraoui and Kapałka redefined opacity [GK10] to require all prefixes of the execution to satisfy the original notion of opacity. By specifying our correctness condition as the set of traces that can be exhibited by an I/O automaton, we ensure *a priori* that the condition is prefix-closed (an automaton cannot generate an execution without first generating all its prefixes). Furthermore, our approach facilitates the construction of hierarchical proofs that are sufficiently precise to be machine-checked. It would be interesting to express the prefix-closed variant of opacity as an automaton, and to formally prove that this automaton implements TMS1 (i.e., that opacity implies TMS1).

As we were working on our TMS1 condition, we became aware of work by Imbs, et al. [IdMR08, IdMR09], who share our basic criticism of opacity, and had proposed a new condition, called *virtual world consistency* (VWC), to address the problem. Like TMS1, and as illustrated in Figure 5, VWC relaxes opacity's requirement to justify all committed and aborted transactions using a single serial execution, simply requiring that aborted ones never observe behaviour that could not have occurred. However, as described below, we believe that VWC goes too far in relaxing this requirement.

The interfaces and assumptions used to define VWC differ in several ways from those of TMS1 and opacity, which leads to significant differences in what conditions can be expressed, and what executions are allowed. Unlike TMS1 and opacity, in which behaviour is modelled by a sequence of events, VWC is defined in a model in which a transaction is a group of operations by a process, and transactions are related by "process order" (transactions executed by the same process are totally ordered) and a "reads-from" relation (one transaction reads a value written by another). To keep the latter relation well-defined, VWC is defined only for the limited case of a read-write memory in which no two writes ever write the same value. Furthermore, VWC allows aborted transactions to ignore committed transactions that precede them in real time. (Indeed, the model has no notion of external order between transactions by different processes.) This appears to derive from an implicit assumption that transactions are the only means of communication. We do not think such an assumption is reasonable in practice in our target context (TM runtimes for unmanaged languages), and

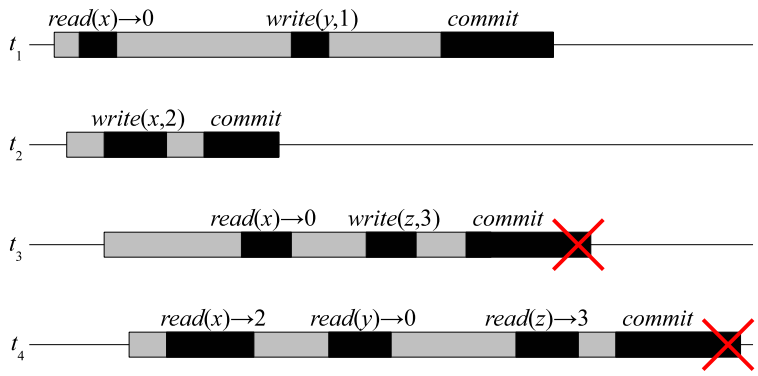

Fig. 6. An execution that is allowed by TMS1 but not by either opacity or VWC: The only possible serialisation of the committed transactions is $t_1 t_2$, which yields the following possibilities for (x, y, z): $(0, 0, 0)$, $(0, 1, 0)$ and $(2, 1, 0)$. But t_4 observes $(x, y, z) = (2, 0, 3)$, so the execution is not opaque. It is not allowed by VWC because no committed transaction writes 3 to z. However, it is allowed by TMS1 because the value returned by every read operation, even of aborted transactions, can be justified. The only nontrivial one is t_4's read of z. In that case, the *validResp* condition is satisfied by choosing $S = \{t_2, t_3\}$: When this response occurs, both of these transactions are visible (i.e., they have invoked commit), and because *extOrder* is empty, $t_3 t_2 t_4$ is trivially a serialisation of an externally consistent prefix set that respects the external order, and $ops(t_3 t_2 t_4) \cdot (read(z), 3)$ is a legal sequential history.

therefore we require executions to satisfy external consistency. Thus, TMS1 excludes some behaviours that VWC allows. Finally, unlike the interfaces of opacity and TMS1, VWC does not model transaction commit (or any other operation) as an interval with an invocation and a response, which prevents it from allowing executions in which a read is justified by a write of a transaction that ultimately does not commit, as both opacity and TMS1 do. Figure 6 illustrates an execution that TMS1 allows, but that satisfies neither opacity nor VWC.

5. TMS2: a stricter, more pragmatic condition

In this section, we introduce the TMS2 automaton, which captures some of the structure common to many practical TM implementations, in particular, those that support read-write memory (rather than the general object semantics supported by TMS1), and use a "deferred update" approach. Such an implementation maintains a private *write set* for each transaction, which records the locations, and the associated values, written by the transaction. A transaction that writes one or more locations and commits (henceforth, a *successful writing transaction*) takes effect at some point between the invocation and response of its commit operation; the writes are "deferred" to this point. In addition, the set of values observed by any transaction—the *read set* of the transaction—is always consistent with the memory state at some point during its execution. Note that this memory state may be any memory state that was current at some point since the transaction began. Thus, a read-only transaction may appear to take effect at some point before it invokes commit. This flexibility is important because some algorithms (e.g., TL2 [DSS06]) exhibit such behaviour. However, to guarantee that committed transactions can be ordered into a serial execution such that concatenating the operations of these transactions in order of the serial execution yields a legal sequential history, the read set of a successful writing transaction must be consistent with the memory state current at the time the transaction takes effect.

Because TMS2 captures the intuition of many TM implementations, we believe that most implementors will find TMS2 easier to understand than TMS1, and that it will often be more straightforward to prove that an implementation satisfies TMS2 than that it satisfies TMS1. In Section 6, we prove that every trace of TMS2 is also a trace of TMS1 (specialized for read-write memory); that is, every behaviour allowed by TMS2 is allowed by TMS1. Thus, proving that an implementation satisfies TMS2 suffices to prove that it satisfies TMS1.

The TMS2 automaton (Figure 7) records the sequence *mem* of memory states produced by successful writing transactions. (The sequence is indexed from 0, so mem_0 is a memory state satisfying the specified initialisation predicate, and mem_k is the memory state written by the kth successful writing transaction.) We

State variables

mem: sequence of functions mapping L to V;
initially one function m such that $init(m)$.
For each $t \in \mathcal{T}$:
pc_t: PCvals; initially notStarted
$beginIdx_t$: $\mathbb{N}$; initially arbitrary
$rdSet_t$: $L \to V_\perp$; initially all $\perp$
$wrSet_t$: $L \to V_\perp$; initially all $\perp$

Auxiliary history variables

$extOrder$: binary relation on $\mathcal{T}$; initially empty
For each $t \in \mathcal{T}$:
ops_t: sequence of operations (i.e., $(\mathcal{I} \times \mathcal{R})^*$);
initially empty
$pendingOp_t$: $\mathcal{I}$; initially arbitrary
$invokedCommit_t$: Boolean; initially **false**
For each $n \in \mathbb{N}$:
T_n: $\mathcal{T}^*$; initially empty

External actions for each $t \in \mathcal{T}$

begin_t
Pre: $pc_t = \mathsf{notStarted}$
Eff: $pc_t \leftarrow \mathsf{beginPending}$
$beginIdx_t \leftarrow |mem| - 1$
$extOrder \leftarrow extOrder \cup (DT \times \{t\})$

$\mathsf{beginOk}_t$
Pre: $pc_t = \mathsf{beginPending}$
Eff: $pc_t \leftarrow \mathsf{ready}$

$\mathsf{inv}_t(read(l))$, $l \in L$
Pre: $pc_t = \mathsf{ready}$
Eff: $pc_t \leftarrow \mathsf{doRead}(l)$
$pendingOp_t \leftarrow read(l)$

$\mathsf{resp}_t(v)$, $v \in V$
Pre: $pc_t = \mathsf{readResp}(v)$
Eff: $pc_t \leftarrow \mathsf{ready}$
$ops_t \leftarrow ops_t \cdot (pendingOp_t, v)$

$\mathsf{inv}_t(write(l, v))$, $l \in L, v \in V$
Pre: $pc_t = \mathsf{ready}$
Eff: $pc_t \leftarrow \mathsf{doWrite}(l, v)$
$pendingOp_t \leftarrow write(l, v)$

$\mathsf{resp}_t(\mathrm{ok})$
Pre: $pc_t = \mathsf{writeRespOk}$
Eff: $pc_t \leftarrow \mathsf{ready}$
$ops_t \leftarrow ops_t \cdot (pendingOp_t, \mathrm{ok})$

commit_t
Pre: $pc_t = \mathsf{ready}$
Eff: $pc_t \leftarrow \mathsf{doCommit}$
$invokedCommit_t \leftarrow \mathbf{true}$

$\mathsf{commitOk}_t$
Pre: $pc_t = \mathsf{commitRespOk}$
Eff: $pc_t \leftarrow \mathsf{committed}$

cancel_t
Pre: $pc_t = \mathsf{ready}$
Eff: $pc_t \leftarrow \mathsf{cancelPending}$

abort_t
Pre: $pc_t \notin \{\mathsf{notStarted}, \mathsf{ready}, \mathsf{commitRespOk}, \mathsf{committed}, \mathsf{aborted}\}$
Eff: $pc_t \leftarrow \mathsf{aborted}$

Internal actions for each $t \in \mathcal{T}$

$\mathsf{doCommitReadOnly}_t(n)$, $n \in \mathbb{N}$
Pre: $pc_t = \mathsf{doCommit}$
$\mathbf{dom}(wrSet_t) = \emptyset$
$validIdx(t, n)$
Eff: $pc_t \leftarrow \mathsf{commitRespOk}$
$T_n \leftarrow T_n \cdot t$

$\mathsf{doCommitWriter}_t$
Pre: $pc_t = \mathsf{doCommit}$
$\mathbf{dom}(wrSet_t) \neq \emptyset$
$readCons(mem_{\mathrm{last}}, rdSet_t)$
Eff: $pc_t \leftarrow \mathsf{commitRespOk}$
$T_n \leftarrow t$, where $n = |mem|$
$mem \leftarrow mem \cdot mem_{\mathrm{last}}[wrSet_t]$

$\mathsf{doRead}_t(l, n)$, $l \in L, n \in \mathbb{N}$
Pre: $pc_t = \mathsf{doRead}(l)$
$l \in \mathbf{dom}(wrSet_t) \vee validIdx(t, n)$
Eff: if $l \in \mathbf{dom}(wrSet_t)$ then
$pc_t \leftarrow \mathsf{readResp}(wrSet_t(l))$
else
$v \leftarrow mem_n(l)$
$pc_t \leftarrow \mathsf{readResp}(v)$
$rdSet_t \leftarrow rdSet_t[l \to v]$

$\mathsf{doWrite}_t(l, v)$, $l \in L, v \in V$
Pre: $pc_t = \mathsf{doWrite}(l, v)$
Eff: $pc_t \leftarrow \mathsf{writeRespOk}$
$wrSet_t \leftarrow wrSet_t[l \to v]$

Functions and predicates

$$\mathsf{PCvals} \triangleq \{\mathsf{notStarted}, \mathsf{beginPending}, \mathsf{ready}, \mathsf{doCommit}, \mathsf{commitRespOk}, \mathsf{cancelPending}, \mathsf{committed}, \mathsf{aborted}\} \cup \{\mathsf{doRead}(l) \mid l \in L\} \cup \{\mathsf{readResp}(v) \mid v \in V\} \cup \{\mathsf{doWrite}(l, v) \mid l \in L \wedge v \in V\} \cup \{\mathsf{writeRespOk}\}$$

$$DT \triangleq \{t \mid pc_t \in \{\mathsf{committed}, \mathsf{aborted}\}\}$$

$$readCons(m, rdSet) \triangleq \forall l \in \mathbf{dom}(rdSet), rdSet(l) = m(l)$$

$$validIdx(t, n) \triangleq beginIdx_t \leq n < |mem| \wedge readCons(mem_n, rdSet_t)$$

Fig. 7. The $\mathsf{TMS2}(init)$ automaton, a TM that supports read-write memory with initialisation predicate $init$.

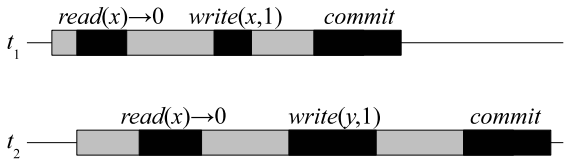

Fig. 8. An opaque execution that is not allowed by TMS2: Because t_2 reads 0 from x, t_2 must be ordered before t_1, which opacity allows. However, t_1's commit operation occurs entirely before t_2's, so TMS2 requires t_1 to commit before t_2. When t_2 executes its doCommitWriter action, its validation will fail as its read set is not consistent with the memory state installed by t_1.

use mem_{last} to denote the last element of *mem*. In addition, for each transaction t, the automaton maintains its read and write sets, $rdSet_t$ and $wrSet_t$, and two bookkeeping variables, pc_t, which keeps track of t's next step, and $beginIdx_t$, which records which memory state is current when t begins. (We ignore for now the history variables of TMS2, which are used only to facilitate the proof that TMS2 implements TMS1. Because they do not occur in the preconditions of any of the actions, nor in the expressions for values assigned to the actual state variables, these history variables do not affect the behaviour of the automaton [OG76].)

The external actions of TMS2, which are just the actions of TMS1 specialized for read-write memory, simply check and update the bookkeeping variables. The "real work" of the operations, checking and updating the memory state and/or the read and write sets, is done by internal actions. In particular, when a writing transaction t commits (see the $\mathsf{doCommitWriter}_t$ action), its read set must be consistent with the current memory state (i.e., $readCons(mem_{\text{last}}, rdSet_t)$), and it changes the memory state to reflect its writes, adding the new memory state to the sequence of memory states (i.e., $mem \leftarrow mem \cdot mem_{\text{last}}[wrSet_t]$).

To ensure that the read set of any transaction t is always consistent with a memory state that was current at some point during t's execution, whenever t reads a value from a location it has not written (see the doRead_t action), t is *validated against* a memory state mem_n (i.e., $validIdx(t, n)$) that is consistent with t's read set (i.e., $readCons(mem_n, rdSet_t)$) and was current at some point since t began (i.e., $beginIdx_t \leq n$). (When a read-only transaction commits, it similarly validates that its read set is consistent with some memory state that was current since it began—see the $\mathsf{doCommitReadOnly}_t$ action.) If t is reading a value from a location it has not written, it returns the value associated with the location by mem_n. If t is reading a value from a location it *has* written (again, see the doRead_t action), it simply returns the value associated with the location in its write set, which is updated by t's write operations (see the $\mathsf{doWrite}_t$ action).

Note that, unlike in TMS1, there is no validation required to abort a transaction in TMS2. This is because a transaction's effects cannot be seen by others until the transaction executes its doCommit action, after which the transaction will receive a commitOk response (provided that it keeps taking steps). We say that such a transaction is *effectively committed*.

Like opacity, TMS2 disallows many executions that would be acceptable for TM, and are allowed by TMS1. For example, none of the executions illustrated in Figures 2, 3, 5 and 6 is allowed by TMS2. However, TMS2 disallows these executions deliberately, in a way that allows it to capture a simpler condition that is closer to intuition about a large class of practical implementations.

We believe, but have not proved, that every execution allowed by TMS2 also satisfies opacity (modulo cosmetic interface differences). If we expressed opacity as automaton, as mentioned in Section 4, we could formally prove this relationship. For now, we simply observe that opacity allows some executions that TMS2 does not, as illustrated by the example in Figure 8.

6. TMS2 implements TMS1

In this section, we prove that TMS2(*init*) implements TMS1(RW(*init*)). Henceforth in this section, we fix *init*, and simply write TMS2 and TMS1 for the automata. We have formalised and verified this proof using the PVS theorem prover system [PVS]. We first describe the key ideas in the proof, and then present formal invariants and lemmas that support it in the following sections.

To prove that TMS2 implements TMS1, we show that the function f in Figure 9, which maps each state of TMS2 to a state of TMS1, is a refinement (see Section 2.3). That is, we prove that any initial state of TMS2 is mapped to an initial state of TMS1, and that for any step of TMS2, there is a corresponding (possibly

f maps states of TMS2(m) to states of TMS1(RW(m)) such that for any state s of TMS2(m),

$$f(s).extOrder = s.extOrder$$

and for all $t \in \mathcal{T}$,

$$\begin{aligned} f(s).status_t &= abs(s.pc_t) \\ f(s).ops_t &= s.ops_t \\ f(s).pendingOp_t &= s.pendingOp_t \\ f(s).invokedCommit_t &= s.invokedCommit_t \end{aligned}$$

where

$$abs(pc) = \begin{cases} \mathsf{opPending} & \text{if } pc \in \{\mathsf{doRead}(l) \mid l \in L\} \cup \{\mathsf{readResp}(v) \mid v \in V\} \\ & \qquad \cup \{\mathsf{doWrite}(l, v) \mid l \in L \wedge v \in V\} \cup \{\mathsf{writeRespOk}\} \\ \mathsf{commitPending} & \text{if } pc \in \{\mathsf{doCommit}, \mathsf{commitRespOk}\} \\ pc & \text{if } pc \in \{\mathsf{notStarted}, \mathsf{beginPending}, \mathsf{ready}, \mathsf{cancelPending}, \mathsf{committed}, \mathsf{aborted}\} \end{cases}$$

Fig. 9. A refinement from TMS2(m) to TMS1(RW(m)).

empty) sequence of steps of TMS1 that preserves the mapping. To enable this, we augment TMS2 with several history variables (shown in Figure 7). The *extOrder*, ops_t, $pendingOp_t$ and $invokedCommit_t$ variables simply maintain the corresponding state variables from TMS1. The T_n history variables record which effectively committed transactions (i.e., transactions that have executed their doCommit action) validate against which memory states (and the order in which they do so). Specifically, the $\mathsf{doCommitReadOnly}_t(n)$ action validates t against mem_n and appends t to T_n, and the $\mathsf{doCommitWriter}_t$ action validates t against mem_{last} and starts a new sequence with just t (i.e., $T_n \leftarrow t$ where $n = |mem|$). Thus, T_0 contains all effectively committed read-only transactions that validated against the initial state mem_0, and for $0 < n < |mem|$, T_n contains the nth effectively committed writing transaction, which writes mem_n, followed by all effectively committed read-only transactions that validate against mem_n. (For $n \geq |mem|$, T_n is empty.)

We abuse terminology by applying terms developed for TMS1 to the corresponding concepts in TMS2. For example, we say that a transaction t is *commit-pending* if $pc_t \in \{\mathsf{doCommit}, \mathsf{commitRespOk}\}$, and that a set S of transactions is an *externally consistent prefix set* if for all $t, t' \in \mathcal{T}$, $t' \in S \wedge (t, t') \in extOrder \implies (t \in S \iff pc_t = \mathsf{committed})$.

The first proof obligation for showing that f is a refinement (the start state condition) is immediate: in start states of both automata, *extOrder* is empty, and for all $t \in \mathcal{T}$, ops_t is empty, $invokedCommit_t$ is **false**, and $pendingOp_t$ can be any operation invocation, and for all $t \in \mathcal{T}$, $pc_t = \mathsf{notStarted}$ in TMS2 and $status_t = \mathsf{notStarted}$ in TMS1.

For the second proof obligation (the step condition), the choice of TMS1 step(s) for a given step of TMS2 is straightforward because TMS1 has no internal actions: For a step with an internal action of TMS2, no step is taken by TMS1; we show that the prestate and the poststate of TMS2 are mapped by the refinement f to the same state of TMS1 (see Lemma 28). For a step with an external action, the same action is taken in TMS1; we show that this action is enabled in the TMS1 state corresponding to the TMS2 prestate, and that state resulting from executing the action in TMS1 corresponds to the TMS2 poststate (see Lemmas 29–37). For most external actions, this is straightforward, but for resp_t, $\mathsf{commitOk}_t$ and abort_t, we must show that the appropriate validation condition holds. In TMS2, the preconditions of these actions simply check the bookkeeping variable pc_t; the actual validation is done by the associated internal action. We use the T_n history variables to prove that the internal actions ensure the appropriate validation condition holds and is not invalidated by subsequent steps of TMS2.

In particular, we prove the following three key properties about the sequence T^* of transactions resulting from concatenating the T_n variables in order (i.e., $T^* = T_0 \cdot \cdots \cdot T_k$, where $k = |mem| - 1$):

- T^* is a serialisation of the effectively committed transactions that respects the external order (see Invariant 18).
- Concatenating the operations of these transactions in order according to T^* yields a legal sequential history (see Invariant 24).
- For any active transaction t, there is some prefix of T^* such that concatenating the operations of trans-

actions in order according to the prefix followed by the operations of t yields a legal sequential history (see Invariants 20 and 25).

Proving these properties entails most of the complexity of the proof.

Because the effectively committed transactions include all committed transactions and some commit-pending ones (and no others), and because a transaction executing commitOk is effectively committed, and one executing abort is not, the first two of these properties imply that T^* can be used to show that the appropriate validation condition is satisfied for these two actions (see Lemma 26). For a resp_t action, the third property guarantees that a prefix of T^* followed by t yields a serialisation that justifies the responses of the transactions in that serialisation, and we verify this serialisation respects the external order (see Invariant 19) and its transactions form an externally consistent prefix set (see Lemma 27), so this serialisation satisfies the required validation condition.

6.1. Invariants and lemmas about TMS2

In this section, we state and prove several invariants and lemmas about TMS2 that are used in the next section to prove that the function f defined in Figure 9 is a refinement. The proofs in these sections are detailed enough to serve as the basis for a fully formal machine-checkable proof, and we have completed such a formal proof using the PVS prover [PVS]. In particular, we explicitly state several trivial lemmas and invariants that are often omitted in informal proofs (and in doing the proof in PVS, we even discovered one—Invariant 12—that we had omitted from our original manual proof on which we based the PVS proof). As required by PVS (and unlike the proof overview above, in which we often describe high-level properties before stating the lower-level properties required to prove them), each lemma and invariant in this section and the next depend only on the lemmas and invariants (and definitions) that precede them.

We first define two derived state variables for TMS2: ECT, which is the set of effectively committed transactions; and $doneOps_t$, which is the sequence of operations for which transaction t has executed an appropriate "do" action. Formally,

$$ECT = \{t \mid pc_t = \mathsf{commitRespOk} \vee pc_t = \mathsf{committed}\}$$

$$doneOps_t = \begin{cases} ops_t \cdot (pendingOp_t, v) & \text{if } pc_t = \mathsf{readResp}(v) \\ ops_t \cdot (pendingOp_t, \mathrm{ok}) & \text{if } pc_t = \mathsf{writeRespOk} \\ ops_t & \text{otherwise} \end{cases}$$

We also use $acts_t$ to denote the set of all actions associated with a transaction t.

The following three lemmas follow immediately by inspection of the TMS2 automaton (Figure 7). Lemmas 7 and 8 assert which actions might change each state variable (including the history and derived variables). Lemma 9 identifies some simple stable properties of the automaton (i.e., properties that, once true, remain true). Although they are trivial to verify, we find it helpful when verifying the rest of the proof to have these lemmas cited explicitly. Indeed, the PVS prover often requires this in the formal proof.

Lemma 7 If $(s, a, s') \in trans(\mathsf{TMS2})$ then

- $s'.extOrder = s.extOrder$ unless $a = \mathsf{begin}_t$ for some t,
- $s'.mem = s.mem$ unless $a = \mathsf{doCommitWriter}_t$ for some t,
- $s'.ECT = s.ECT$ unless $a = \mathsf{doCommitReadOnly}_t(n)$ for some t and n or $a = \mathsf{doCommitWriter}_t$ for some t,

and for all $t \in \mathcal{T}$,

- $s'.beginIdx_t = s.beginIdx_t$ unless $a = \mathsf{begin}_t$,
- $s'.invokedCommit_t = s.invokedCommit_t$ unless $a = \mathsf{commit}_t$,
- $s'.pendingOp_t = s.pendingOp_t$ unless $a = \mathsf{inv}_t(read(l))$ for some l or $a = \mathsf{inv}_t(write(l, v))$ for some l and v,
- $s'.rdSet_t = s.rdSet_t$ unless $a = \mathsf{doRead}_t(l, n)$ for some l and n and $l \notin \mathbf{dom}(s.wrSet_t)$,
- $s'.wrSet_t = s.wrSet_t$ unless $a = \mathsf{doWrite}_t(l, v)$ for some l and v,
- $s'.ops_t = s.ops_t$ unless $a = \mathsf{resp}_t(r)$ for some r,
- $s'.doneOps_t = s.doneOps_t$ unless $a = \mathsf{doRead}_t(l, n)$ for some l and n or $a = \mathsf{doWrite}_t(l, v)$ for some l and v,

and for all $n \in \mathbb{N}$,

- $s'.T_n = s.T_n$ unless $a = \mathsf{doCommitReadOnly}_t(n)$ for some t or $a = \mathsf{doCommitWriter}_t$ for some t and $n = |s.mem|$.

Lemma 8 If $(s, a, s') \in \mathit{trans}(\mathsf{TMS2})$ and $a \notin \mathit{acts}_t$ then $s'.pc_t = s.pc_t$.

Lemma 9 If $(s, a, s') \in \mathit{trans}(\mathsf{TMS2})$ then for all $t \in \mathcal{T}$

- $s.pc_t \neq \mathsf{notStarted} \implies s'.pc_t \neq \mathsf{notStarted}$
- $s.pc_t = \mathsf{committed} \implies s'.pc_t = \mathsf{committed}$
- $s.pc_t = \mathsf{aborted} \implies s'.pc_t = \mathsf{aborted}$
- $s.\mathit{invokedCommit}_t \implies s'.\mathit{invokedCommit}_t$

and $s.\mathit{extOrder} \subseteq s'.\mathit{extOrder}$, $|s.mem| \leq |s'.mem|$, and $s'.mem_n = s.mem_n$ for $n \in [0, |s.mem| - 1]$.

The following invariants are easily verified by induction using the lemmas above and simple observations about the transition relation and initialisation conditions of TMS2. Invariant 10 says that there is a first memory state, and it satisfies the initialisation predicate. Invariant 11 says that, before a transaction starts, its read set is empty. Invariant 12 says that the "begin index" of a transaction that has started is in the domain of *mem*. Invariant 13 says that the operation a transaction t is about to perform is recorded in $\mathit{pendingOp}_t$. Invariant 14 says that a transaction that is effectively committed, or has invoked commit but not yet executed its doCommit action, is visible. Invariant 15 says that if one transaction is ordered before another by the external order, then the first transaction has completed and the second transaction has started.

Invariant 10 $|mem| > 0$ and mem_0 satisfies the initialisation predicate (i.e., $\mathit{init}(mem_0)$).

Invariant 11 If $pc_t = \mathsf{notStarted}$ then $\mathbf{dom}(\mathit{rdSet}_t) = \emptyset$.

Invariant 12 If $pc_t \neq \mathsf{notStarted}$ then $\mathit{beginIdx}_t < |mem|$.

Invariant 13 For $t \in \mathcal{T}$:

- If $pc_t = \mathsf{doRead}(l)$ then $\mathit{pendingOp}_t = \mathit{read}(l)$.
- If $pc_t = \mathsf{doWrite}(l, v)$ then $\mathit{pendingOp}_t = \mathit{write}(l, v)$.

Invariant 14 If $pc_t = \mathsf{doCommit}$ or $t \in \mathit{ECT}$ then $\mathit{invokedCommit}_t$.

Invariant 15 If $(t, t') \in \mathit{extOrder}$ then $t \in \mathit{DT}$ and $pc_{t'} \neq \mathsf{notStarted}$.

We now show the first of the key properties from the proof overview: that concatenating the T_n variables for $0 \leq n < |mem|$ in order yields a serialisation of the effectively committed transactions that respects the external order (Invariant 18), from which it immediately follows that a prefix of this sequence is a serialisation that respects the external order (Invariant 19). To do so, we first show that the transactions in the T_n variables are exactly the effectively committed transactions (Invariant 16), and that if two transactions are ordered by real time and the first is effectively committed, then the first transaction validates against a state that was written before the second transaction began (Invariant 17). As a shorthand, we write $T_{j..k}$ for $T_j \cdot \cdots \cdot T_k$, with the usual convention that $T_{j..k}$ is empty if $j > k$.

Invariant 16 $\mathit{ECT} = \{t \mid t \text{ in } T_k \text{ for some } k < |mem|\}$

Proof. By induction on the length of an execution. In any start state of TMS2, the invariant holds because $\mathit{ECT} = \emptyset$ and T_k is empty for all k. Suppose the invariant holds in a reachable state s and that $(s, a, s') \in \mathit{trans}(\mathsf{TMS2})$. By Lemma 7, $s'.\mathit{ECT} = s.\mathit{ECT}$, $s'.mem = s.mem$, and $s'.T_k = s.T_k$ for all k, so the invariant holds in s' by the inductive hypothesis, unless $a = \mathsf{doCommitReadOnly}_t(n)$ for some t and n or $a = \mathsf{doCommitWriter}_t$ for some t.

If $a = \mathsf{doCommitReadOnly}_t(n)$ then $n < |s.mem| = |s'.mem|$, $s'.ECT = s.ECT \cup \{t\}$, $s'.T_n = s.T_n \cdot t$, and $s'.T_k = s.T_k$ for all $k \neq n$, so the invariant holds in s' by the inductive hypothesis.

If $a = \mathsf{doCommitWriter}_t$ then $s'.ECT = s.ECT \cup \{t\}$, $s'.T_n = t$, and $s'.T_k = s.T_k$ for all $k \neq n$, where $n = |s.mem| = |s'.mem| - 1$, so the invariant holds in s' by the inductive hypothesis. □

Invariant 17 If $(t, t') \in extOrder$ and $t \in ECT$ then t is in T_k for some $k \leq beginIdx_{t'}$.

Proof. By induction on the length of an execution. In any start state of TMS2, the invariant holds trivially because $ECT = \emptyset$. Suppose the invariant holds in a reachable state s and that $(s, a, s') \in trans(\mathsf{TMS2})$.

If $a = \mathsf{begin}_{t'}$ then either $t \notin s'.ECT$, in which case the invariant holds trivially in s', or by Invariant 16 (applied to s'), t is in $s'.T_k$ for some $k \leq |s.mem| - 1 = s'.beginIdx_{t'}$.

Otherwise, by Lemma 7, $s'.beginIdx_{t'} = s.beginIdx_{t'}$. If $(t, t') \notin s.extOrder$ then by Lemma 7 and the effect of the begin action, $(t, t') \notin s'.extOrder$, and so the invariant holds trivially in s'. If $(t, t') \in s.extOrder$ then by Invariant 15, $s.pc_{t'} \neq \mathsf{notStarted}$ and $t \in s.DT$, so $s.pc_t = \mathsf{aborted}$ or $s.pc_t = \mathsf{committed}$. In the first case, $s'.pc_t = \mathsf{aborted}$ by Lemma 9, so the invariant holds trivially in s'. In the second case, $t \in s.ECT$, so by the inductive hypothesis, t is in $s.T_k$ for some $k \leq s.beginIdx_{t'} = s'.beginIdx_{t'}$. By Lemma 7, $s'.T_k = s.T_k$, and so the invariant holds in s', unless $a = \mathsf{doCommitReadOnly}_{t''}(k)$ for some t'', in which case t is in $s'.T_k = s.T_k \cdot t''$, or $a = \mathsf{doCommitWriter}_{t''}$ for some t'' and $k = |s.mem|$, which is impossible, since $k \leq s.beginIdx_{t'} < |s.mem|$ by Invariant 12. □

Invariant 18 $T_{0..k} \in ser(ECT, extOrder)$, where $k = |mem| - 1$.

Proof. By induction on the length of an execution. In any start state of TMS2, the invariant holds because $ECT = \emptyset$, $|mem| = 1$ and T_0 is empty. Suppose the invariant holds in a reachable state s and that $(s, a, s') \in trans(\mathsf{TMS2})$. By Lemma 7, $s'.extOrder = s.extOrder$, $s'.mem = s.mem$, $s'.ECT = s.ECT$ and $s'.T_j = s.T_j$ for all j, so the invariant holds in s' by the inductive hypothesis, unless $a = \mathsf{begin}_t$ for some t, $a = \mathsf{doCommitReadOnly}_t(n)$ for some t and n, or $a = \mathsf{doCommitWriter}_t$ for some t.

If $a = \mathsf{begin}_t$ then $s'.pc_t = \mathsf{beginPending}$, $s'.extOrder = s.extOrder \cup (s.DT \times \{t\})$, $s'.mem = s.mem$, $s'.ECT = s.ECT$, and $s'.T_j = s.T_j$ for all j. Thus, $t \notin s'.ECT$, so $s'.extOrder \cap (s'.ECT \times s'.ECT) = s.extOrder \cap (s.ECT \times s.ECT)$, and the invariant holds in s' by the inductive hypothesis.

If $a = \mathsf{doCommitReadOnly}_t(n)$ then $s.beginIdx_t \leq n < |s.mem| = |s'.mem|$, $s.pc_t = \mathsf{doCommit}$, $s'.pc_t = \mathsf{commitRespOk}$, $s'.extOrder = s.extOrder$, $s'.ECT = s.ECT \cup \{t\}$, $s'.T_n = s.T_n \cdot t$, and $s'.T_j = s.T_j$ for all $j \neq n$. Let $k = |s.mem| - 1 = |s'.mem| - 1$, $\sigma = s.T_{0..k}$, and $\sigma' = s'.T_{0..k} = s.T_{0..n} \cdot t \cdot s.T_{n+1..k}$. By the inductive hypothesis, σ is a serialisation of $s.ECT$, so σ' is a serialisation of $s.ECT \cup \{t\} = s'.ECT$ (since $t \notin s.ECT$). If $(t', t'') \in s'.extOrder \cap (s'.ECT \times s'.ECT)$ then by Invariant 15 (applied to s'), $t' \neq t$ (since $t \notin s'.DT$), so either $(t', t'') \in s.extOrder \cap (s.ECT \times s.ECT)$, in which case $t' \leq_\sigma t''$ by the inductive hypothesis, or $t'' = t$, in which case, t' is in $s.T_j$ for some $j \leq s.beginIdx_t \leq n$ by Invariant 17. So in either case, $t' \leq_{\sigma'} t''$, and thus $\sigma' \in ser(s'.ECT, s'.extOrder)$.

If $a = \mathsf{doCommitWriter}_t$ then $s.pc_t = \mathsf{doCommit}$, $s'.pc_t = \mathsf{commitRespOk}$, $s'.extOrder = s.extOrder$, $s'.ECT = s.ECT \cup \{t\}$, and $s'.T_k = t$ and $s'.T_j = s.T_j$ for all $j \neq k$, where $k = |s.mem| = |s'.mem| - 1$. Let $\sigma = s.T_{0..k-1}$ and $\sigma' = s'.T_{0..k} = s.T_{0..k-1} \cdot t = \sigma \cdot t$. By the inductive hypothesis, σ is a serialisation of $s.ECT$, so σ' is a serialisation of $s.ECT \cup \{t\} = s'.ECT$ (since $t \notin s.ECT$). If $(t', t'') \in s'.extOrder \cap (s'.ECT \times s'.ECT)$ then by Invariant 15 (applied to s'), $t' \neq t$ (since $t \notin s'.DT$), so t' is in σ by the inductive hypothesis (since $t' \in s'.ECT \setminus \{t\} = s.ECT$), and either $t'' = t$, or $(t', t'') \in s.extOrder \cap (s.ECT \times s.ECT)$, in which case $t' \leq_\sigma t''$ by the inductive hypothesis. In either case, $t' \leq_{\sigma'} t''$, and thus $\sigma' \in ser(s'.ECT, s'.extOrder)$. □

Invariant 19 If $n < |mem|$ and $S = \{t \mid t \text{ in } T_k \text{ for some } k \leq n\}$ then $S \subseteq ECT$ and $T_{0..n} \in ser(S, extOrder)$.

Proof. Immediate from Invariant 18. □

Invariant 20 says that for every transaction that has started, there is some memory state that is consistent with the transaction's read set and was current at some time after the transaction began. This holds because whenever a location is added to a read set (i.e., in one case of the doRead action), it is associated with the value of the location in some memory state that is already consistent with the transaction's read set.

Invariant 20 If $pc_t \neq \mathsf{notStarted}$ then $validIdx(t, n)$ for some n.

Proof. By induction on the length of an execution. In any start state of TMS2, this invariant holds because $pc_t = \mathsf{notStarted}$. Suppose the invariant holds in a reachable state s and that $(s, a, s') \in \mathit{trans}(\mathsf{TMS2})$.

If $s.pc_t = \mathsf{notStarted}$ then $s'.pc_t = \mathsf{notStarted}$, and so the invariant holds trivially in s', unless $a = \mathsf{begin}_t$. In this case, $\mathbf{dom}(s'.rdSet_t) = \mathbf{dom}(s.rdSet_t) = \emptyset$ by Invariant 11, so $\mathit{readCons}(m, s'.rdSet_t)$ for any m. Thus, $s'.\mathit{validIdx}(t, n)$ where $n = s'.\mathit{beginIdx}_t = |s.mem| - 1 = |s'.mem| - 1$, as required.

If $s.pc_t \neq \mathsf{notStarted}$ then $a \neq \mathsf{begin}_t$ and by the inductive hypothesis, $s.\mathit{validIdx}(t, n)$ for some n, and thus, $s.\mathit{beginIdx}_t \leq n < |s.mem|$ and $\mathit{readCons}(s.mem_n, s.rdSet_t)$. By Lemma 7, $s'.\mathit{beginIdx}_t = s.\mathit{beginIdx}_t$, and by Lemma 9, $|s'.mem| \geq |s.mem| > n$ and $s'.mem_n = s.mem_n$. By Lemma 7 again, $s'.rdSet_t = s.rdSet_t$, in which case $\mathit{readCons}(s'.mem_n, s'.rdSet_t)$ and thus $s'.\mathit{validIdx}(t, n)$, unless $a = \mathsf{doRead}_t(l, n')$ for some $l \notin \mathbf{dom}(wrSet_t)$ and n'. In that case, by the precondition of $\mathsf{doRead}_t(l, n')$, $s.\mathit{validIdx}(t, n')$, and thus $s'.\mathit{beginIdx}_t = s.\mathit{beginIdx}_t \leq n' < |s.mem| = |s'.mem|$ and since $s'.rdSet_t = s.rdSet_t[l \to s.mem_{n'}(l)]$, we have $\mathit{readCons}(s'.mem_{n'}, s'.rdSet_t)$, so $s'.\mathit{validIdx}(t, n')$. □

Invariant 21 says that a location is in the write set of a transaction if and only if the transaction has written that location, in which case, the write set associates the location with the last value so written. This holds because whenever a transaction writes a location (i.e., executes its doWrite action), it updates its write set to associate the location with the value written.

Invariant 21 $wrSet_t = \mathit{writes}(\mathit{doneOps}_t)$.

Proof. By induction on the length of an execution. In any start state of TMS2, this invariant holds because $\mathbf{dom}(wrSet_t) = \emptyset$ and $\mathit{doneOps}_t$ is empty. Suppose the invariant holds in a reachable state s and that $(s, a, s') \in \mathit{trans}(\mathsf{TMS2})$. By Lemma 7, $s'.wrSet_t = s.wrSet_t$ and $s'.\mathit{doneOps}_t = s.\mathit{doneOps}_t$ unless a is one of $\mathsf{doRead}_t(l, n)$ for some l and n or $\mathsf{doWrite}_t(l, v)$ for some l and v.

If $a = \mathsf{doRead}_t(l, n)$ then $s'.wrSet_t = s.wrSet_t$ and by Invariant 13, $s'.\mathit{doneOps}_t = s.\mathit{doneOps}_t \cdot (\mathit{read}(l), v)$ for some $v \in V$. Thus, $\mathit{writes}(s'.\mathit{doneOps}_t) = \mathit{writes}(s.\mathit{doneOps}_t)$, so the invariant holds in s' by the inductive hypothesis.

If $a = \mathsf{doWrite}_t(l, v)$ then $s'.wrSet_t = s.wrSet_t[l \to v]$ and by Invariant 13, $s'.\mathit{doneOps}_t = s.\mathit{doneOps}_t \cdot (\mathit{write}(l, v), \mathrm{ok})$. Thus, $\mathit{writes}(s'.\mathit{doneOps}_t) = \mathit{writes}(s.\mathit{doneOps}_t)[l \to v]$, so the invariant holds in s' by the inductive hypothesis. □

Invariant 22 says that applying the operations of a transaction starting from any memory state that is consistent with the transaction's read set yields the appropriate responses, and leaves the memory so that the locations in the transaction's write set have the values specified by the write set and all other locations are unchanged. It follows straightforwardly by induction because, by Invariant 21, a *read* operation gets the last value written to that location by the transaction, or the value in the memory state from which the transaction starts if the location is not written by the transaction.

Invariant 22 If $\mathit{readCons}(m, rdSet_t)$ then $m \xrightarrow{\mathit{doneOps}_t} m[wrSet_t]$.

Proof. That the result of applying $\mathit{doneOps}_t$ to m is $m[wrSet_t]$ is an immediate consequence of Lemma 6 and Invariant 21.

We prove that $\mathit{readCons}(m, rdSet_t)$ implies that $\mathit{doneOps}_t$ is legal starting from m by induction on the length of an execution. In any start state of TMS2, this invariant holds because $\mathit{doneOps}_t$ is empty, and so is legal from any memory state. Suppose the invariant holds in a reachable state s and that $(s, a, s') \in \mathit{trans}(\mathsf{TMS2})$. By Lemma 7, unless $a = \mathsf{doRead}_t(l, n)$ for some l and n or $a = \mathsf{doWrite}_t(l, v)$ for some l and v, $s'.rdSet_t = s.rdSet_t$, $s'.wrSet_t = s.wrSet_t$ and $s'.\mathit{doneOps}_t = s.\mathit{doneOps}_t$, in which case, the invariant holds in s' by the inductive hypothesis.

If $a = \mathsf{doWrite}_t(l, v)$ then $s'.rdSet_t = s.rdSet_t$ and by Invariant 13, $s'.\mathit{doneOps}_t = s.\mathit{doneOps}_t \cdot (\mathit{write}(l, v), \mathrm{ok})$. Since $s.\mathit{doneOps}_t$ is legal starting from m by the inductive hypothesis, and $(\mathit{write}(l, v), \mathrm{ok})$ is legal starting from any memory state, by Lemma 4, $s'.\mathit{doneOps}_t$ is legal starting from m, so the invariant holds in s'.

If $a = \mathsf{doRead}_t(l, n)$ and $l \in \mathbf{dom}(s.wrSet_t)$ then $s'.rdSet_t = s.rdSet_t$ and by Invariant 13, $s'.\mathit{doneOps}_t = s.\mathit{doneOps}_t \cdot (\mathit{read}(l), v)$, where $v = s.wrSet_t(l)$. Thus, $(\mathit{read}(l), v)$ is legal starting from $m[wrSet_t]$. By the inductive hypothesis, $m \xrightarrow{s.\mathit{doneOps}_t} m[wrSet_t]$, so by Lemma 4 and Invariant 21, $s'.\mathit{doneOps}_t$ is legal starting from m.

If $a = \mathsf{doRead}_t(l, n)$ and $l \notin \mathbf{dom}(s.wrSet_t)$ then $s'.rdSet_t = s.rdSet_t[l \to v]$ and by Invariant 13, $s'.\mathit{doneOps}_t =$

$s.doneOps_t \cdot (read(l), v)$, where $v = s.mem_n(l)$. Suppose $readCons(m, s'.rdSet_t)$. Then $m(l) = v$. If $l \in \mathbf{dom}(s.rdSet_t)$ then because $s.validIdx(t, n)$ by the precondition of $\mathsf{doRead}_t(l, n)$, $s.rdSet_t(l) = s.mem_n(l) = v$, so $s'.rdSet_t = s.rdSet_t$. Otherwise, $l \notin \mathbf{dom}(s.rdSet_t)$, so $\mathbf{dom}(s.rdSet_t) = \mathbf{dom}(s'.rdSet_t) \setminus \{l\}$, and for all $l' \in \mathbf{dom}(s.rdSet_t)$, $s.rdSet_t(l') = s'.rdSet_t(l')$. So in either case, $readCons(m, s.rdSet_t)$. Thus, by the inductive hypothesis, $m \xrightarrow{s.doneOps_t} m[s.wrSet_t]$. And since $l \notin \mathbf{dom}(s.wrSet_t)$, $m[s.wrSet_t](l) = m(l) = v$, so $(read(l), v)$ is legal starting from $m[s.wrSet_t]$. Thus, by Lemma 4, $s'.doneOps_t$ is legal starting from m. □

We now come to a key invariant for proving that TMS2 implements TMS1: Invariant 23 says that sequentially applying the operations of the transactions in $T_{0..k}$ in that order for any k in the domain of mem yields the appropriate responses and results in mem_k. From this invariant follow Invariant 24, which makes precise the second key property from the proof overview, and Invariant 25, which, together with Invariant 20, makes precise the third key property.

Invariant 23 For all $k < |mem|$, $mem_0 \xrightarrow{ops(T_{0..k})} mem_k$.

Proof. By induction on the length of an execution. In any start state of TMS2, the invariant holds because $|mem| = 1$ and T_0 is empty (the empty sequence applied to any memory state legally results in the same memory state). Suppose the invariant holds in a reachable state s and that $(s, a, s') \in trans(\mathsf{TMS2})$.

By Lemma 7 and the preconditions of $\mathsf{resp}_t(v)$ and $\mathsf{resp}_t(\mathrm{ok})$, $s'.ops_t = s.ops_t$ for all $t \in s.ECT$, so by Invariant 16, $s'.ops(s.T_k) = s.ops(s.T_k)$ for all $k < |s.mem|$. By Lemma 7, $s'.mem = s.mem$ and $s'.T_k = s.T_k$ for all k, so the invariant holds in s' by the inductive hypothesis, unless $a = \mathsf{doCommitWriter}_t$ for some t or $a = \mathsf{doCommitReadOnly}_t(n)$ for some t and n.

If $a = \mathsf{doCommitWriter}_t$ then $s'.mem = s.mem \cdot s.mem_{\text{last}}[s.wrSet_t]$, $s'.T_n = t$, and $s'.T_k = s.T_k$ for all $k \neq n$, where $n = |s.mem| = |s'.mem| - 1$. By the inductive hypothesis, for $k < n$, since $s'.ops(s'.T_{0..k}) = s.ops(s.T_{0..k})$, we have $s'.mem_0 = s.mem_0 \xrightarrow{s'.ops(s'.T_{0..k})} s.mem_k = s'.mem_k$. For $k = n$, $s'.ops(s'.T_{0..k}) = s.ops(s.T_{0..n-1}) \cdot s.ops_t$, and by the inductive hypothesis, $s'.mem_0 \xrightarrow{s.ops(s.T_{0..n-1})} s.mem_{\text{last}} = s'.mem_{n-1}$. By the precondition of $\mathsf{doCommitWriter}_t$, $readCons(s.mem_{\text{last}}, s.rdSet_t)$, so by Invariant 22, since $s.ops_t = s.doneOps_t$, we have $s.mem_{\text{last}} \xrightarrow{s.ops_t} s.mem_{\text{last}}[s.wrSet_t] = s'.mem_{\text{last}}$. Thus, by Lemma 4, $s'.mem_0 \xrightarrow{s'.ops(s'.T_{0..k})} s'.mem_{\text{last}} = s'.mem_k$.

If $a = \mathsf{doCommitReadOnly}_t(n)$ then $\mathbf{dom}(s.wrSet_t) = \emptyset$, $n < |s.mem|$, $readCons(s.mem_n, s.rdSet_t)$, $s'.mem = s.mem$, $s'.T_n = s.T_n \cdot t$, and $s'.T_k = s.T_k$ for all $k \neq n$.

- For $k < n$, $s'.ops(s'.T_{0..k}) = s.ops(s.T_{0..k})$, so by the inductive hypothesis, $s'.mem_0 = s.mem_0 \xrightarrow{s'.ops(s'.T_{0..k})} s.mem_k = s'.mem_k$.
- For $k = n$, $s'.ops(s'.T_{0..k}) = s.ops(s.T_{0..n}) \cdot s.ops_t$, and by the inductive hypothesis, $s.mem_0 \xrightarrow{s.ops(s.T_{0..n})} s.mem_n$. Since $readCons(s.mem_n, s.rdSet_t)$ and $s.ops_t = s.doneOps_t$, by Invariant 22, we have $s.mem_n \xrightarrow{s.ops_t} s.mem_n[s.wrSet_t] = s.mem_n$ because $\mathbf{dom}(s.wrSet_t) = \emptyset$. Thus, by Lemma 4, $s'.mem_0 \xrightarrow{s'.ops(s'.T_{0..k})} s.mem_n = s'.mem_k$.
- For $k > n$, if $k < |s'.mem| = |s.mem|$ then $s'.ops(s'.T_{0..k}) = s.ops(s.T_{0..n}) \cdot s.ops_t \cdot s.ops(s.T_{n+1..k})$ and $s.ops(s.T_{0..k}) = s.ops(s.T_{0..n}) \cdot s.ops(s.T_{n+1..k})$. By the inductive hypothesis, $s.mem_0 \xrightarrow{s.ops(s.T_{0..k})} s.mem_k$, and $s.mem_0 \xrightarrow{s.ops(s.T_{0..n})} s.mem_n$, so by Lemma 5, $s.mem_n \xrightarrow{s.ops(s.T_{n+1..k})} s.mem_k$. By the previous case, $s'.mem_0 \xrightarrow{s'.ops(s'.T_{0..n})} s.mem_n$, and since $s'.ops(s'.T_{0..k}) = s'.ops(s'.T_{0..n}) \cdot s.ops(s.T_{n+1..k})$, by Lemma 4, $s'.mem_0 \xrightarrow{s'.ops(s'.T_{0..k})} s'.mem_k$.

□

Invariant 24 $ops(T_{0..k})$ is a legal sequential history, where $k = |mem| - 1$.

Proof. By Invariant 23, $ops(T_{0..k})$ is legal starting from mem_0, so by Invariant 10 and Lemma 3, it is a legal sequential history. □

Invariant 25 If $validIdx(t, n)$ then $ops(T_{0..n}) \cdot doneOps_t$ is a legal sequential history.

Proof. Since $validIdx(t, n)$, we have $readCons(mem_n, rdSet_t)$ and $n < |mem|$, so by Invariant 22, $doneOps_t$ is legal starting from mem_n, and by Invariant 23, $mem_0 \xrightarrow{ops(T_{0..n})} mem_n$. Thus, by Lemma 4, $ops(T_{0..n}) \cdot doneOps_t$ is legal starting from mem_0, so by Invariant 10 and Lemma 3, it is a legal sequential history. □

6.2. Proof that f is a refinement

We now prove several properties about the function f defined in Figure 9. The first two are actually also invariants of TMS2 that help establish the validation conditions required by TMS1.

Lemma 26 says that if a transaction has effectively committed but not yet returned from the commit operation in TMS2, then in the corresponding state of TMS1, it satisfies the validation condition for committing; otherwise, it satisfies the validation condition of aborting.

Lemma 26 If s is a reachable state of TMS2 then for all t, $f(s).validCommit(t)$ holds if $s.pc_t = \mathsf{commitRespOk}$ and $f(s).validFail(t)$ holds if $s.pc_t \neq \mathsf{commitRespOk}$.

Proof. Let $S = \{t \mid s.pc = \mathsf{commitRespOk}\}$ and $\sigma = s.T_{0..k}$, where $k = |s.mem| - 1$. By the definitions of f, CPT, CT and ECT, $S \subseteq f(s).CPT$. and $s.ECT = S \cup f(s).CT$. By Invariants 18 and 24 and the definition of f, $\sigma \in ser(s.ECT, s.extOrder) = ser(S \cup f(s).CT, f(s).extOrder)$ and $s.ops(\sigma) = f(s).ops(\sigma)$ is a legal sequential history. Thus, $f(s).validCommit(t)$ for $t \in S$ and $f(s).validFail(t)$ for $t \notin S$. □

Lemma 27 says that, for any active transaction (and indeed, any non-aborted transaction) and any memory state that was current at some point since the transaction began, the set of effectively committed transactions that validated against that memory state or an earlier one is an externally consistent prefix set.

Lemma 27 If s is a reachable state of TMS2 such that $s.pc_t \neq \mathsf{aborted}$ and $s.beginIdx_t \leq n < |s.mem|$ and $S = \{t' | t' \text{ in } s.T_k \text{ for some } k \leq n\}$ then $S \subseteq f(s).VT$ and $f(s).extConsPrefix(S \cup \{t\})$.

Proof. By Invariant 19, $S \subseteq s.ECT$ and by Invariant 14 and the definition of f, $s.invokedCommit_{t'} = f(s).invokedCommit_{t'} = \mathbf{true}$ for all $t' \in s.ECT$, so $S \subseteq f(s).VT$.

We show that if $(t', t'') \in s.extOrder$ and $t'' \in S \cup \{t\}$ then $t' \in S \cup \{t\} \iff s.pc_{t'} = \mathsf{committed}$ by considering four cases:

- If $t' = t$ then by Invariant 15, $t' \in s.DT$, so $s.pc_{t'} = s.pc_t = \mathsf{committed}$ (since $s.pc_t \neq \mathsf{aborted}$).
- If $t' \in S$ then by Invariant 15, $t' \in s.DT$, so $s.pc_{t'} = \mathsf{committed}$ (since $S \subseteq s.ECT$).
- If $s.pc_{t'} = \mathsf{committed}$ and $t'' = t$ then $t' \in s.ECT$, so by Invariant 17, t' is in $s.T_k$ for some $k \leq s.beginIdx_t$. Since $s.beginIdx_t \leq n$, $t' \in S$.
- If $s.pc_{t'} = \mathsf{committed}$ and $t'' \in S$ then $t' \in s.ECT$ and by Invariant 18, $t' \leq_\sigma t''$ where $\sigma = s.T_{0..k}$ for $k = |s.mem| - 1$. Since t'' is in $s.T_j$ for some $j \leq n$, we have t' in $s.T_{j'}$ for some $j' \leq j$, so $t' \in S$.

By the definition of f, $f(s).extOrder = s.extOrder$ and $f(s).status_{t'} = \mathsf{committed} \iff s.pc_{t'} = \mathsf{committed}$, so $f(s).extConsPrefix(S \cup \{t\})$. □

We now prove several lemmas that establish the step condition for showing that f is a refinement. The only nontrivial part to prove for these lemmas is that for a resp, commitOk or abort action, we must show that the appropriate validation condition is satisfied by the TMS1 state corresponding to the TMS2 prestate (see Lemmas 33, 35 and 37).

Lemma 28 If $(s, a, s') \in trans(\mathsf{TMS2})$ and a is an internal action then $f(s') = f(s)$.

Proof. Since a is an internal action, by Lemma 7, $s'.extOrder = s.extOrder$, and for all $t \in \mathcal{T}$,

- $s'.ops_t = s.ops_t$,
- $s'.pendingOp_t = s.pendingOp_t$, and
- $s'.invokedCommit_t = s.invokedCommit_t$.

If $a \in acts_t$ then by Lemma 8, $f(s').status_{t'} = f(s).status_{t'}$ for all $t' \neq t$, so it only remains to verify that $f(s').status_t = f(s).status_t$:

- If $a = \mathsf{doRead}_t(l, n)$ then $s.pc_t = \mathsf{doRead}(l)$ and $s'.pc_t = \mathsf{readResp}(v)$ for some v, so $f(s).status_t = f(s').status_t = \mathsf{opPending}$.
- If $a = \mathsf{doWrite}_t(l, v)$ then $s.pc_t = \mathsf{doWrite}(l, v)$ and $s'.pc_t = \mathsf{writeRespOk}$, so $f(s).status_t = f(s').status_t = \mathsf{opPending}$.
- If $a = \mathsf{doCommitReadOnly}_t(n)$ or $a = \mathsf{doCommitWriter}_t$ then $s.pc_t = \mathsf{doCommit}$ and $s'.pc_t = \mathsf{commitRespOk}$, so $f(s).status_t = f(s').status_t = \mathsf{commitPending}$.

Thus, in every case $f(s).status_t = f(s').status_t$, so $f(s) = f(s')$. □

Lemma 29 If $(s, \mathsf{begin}_t, s') \in trans(\mathsf{TMS2})$ then $(f(s), \mathsf{begin}_t, f(s')) \in trans(\mathsf{TMS1})$.

Proof. begin_t is enabled in $f(s)$ since by the definition of f, $f(s).status_t = s.pc_t = \mathsf{notStarted}$.
By the definition of f and the effect of begin_t in TMS2, we have:

- $f(s').extOrder = s'.extOrder = s.extOrder \cup (s.DT \times \{t\}) = f(s).extOrder \cup (f(s).DT \times \{t\})$.
- $f(s').status_t = abs(s'.pc_t) = \mathsf{beginPending}$.
- $f(s').status_{t'} = abs(s'.pc_{t'}) = abs(s.pc_{t'}) = f(s).status_{t'}$ for all $t' \neq t$.
- $f(s').ops_{t'} = s'.ops_{t'} = s.ops_{t'} = f(s).ops_{t'}$ for all $t' \in \mathcal{T}$.
- $f(s').pendingOp_{t'} = s'.pendingOp_{t'} = s.pendingOp_{t'} = f(s).pendingOp_{t'}$ for all $t' \in \mathcal{T}$.
- $f(s').invokedCommit_{t'} = s'.invokedCommit_{t'} = s.invokedCommit_{t'} = f(s).invokedCommit_{t'}$ for all $t' \in \mathcal{T}$.

Therefore, by the effect of begin_t in TMS1, we have $(f(s), \mathsf{begin}_t, f(s')) \in trans(\mathsf{TMS1})$. □

Lemma 30 If $(s, \mathsf{beginOk}_t, s') \in trans(\mathsf{TMS2})$ then $(f(s), \mathsf{beginOk}_t, f(s')) \in trans(\mathsf{TMS1})$.

Proof. $\mathsf{beginOk}_t$ is enabled in $f(s)$ since by the definition of f, $f(s).status_t = s.pc_t = \mathsf{beginPending}$.
By the definition of f and the effect of $\mathsf{beginOk}_t$ in TMS2, we have:

- $f(s').extOrder = s'.extOrder = s.extOrder = f(s).extOrder$.
- $f(s').status_t = abs(s'.pc_t) = \mathsf{ready}$.
- $f(s').status_{t'} = abs(s'.pc_{t'}) = abs(s.pc_{t'}) = f(s).status_{t'}$ for all $t' \neq t$.
- $f(s').ops_{t'} = s'.ops_{t'} = s.ops_{t'} = f(s).ops_{t'}$ for all $t' \in \mathcal{T}$.
- $f(s').pendingOp_{t'} = s'.pendingOp_{t'} = s.pendingOp_{t'} = f(s).pendingOp_{t'}$ for all $t' \in \mathcal{T}$.
- $f(s').invokedCommit_{t'} = s'.invokedCommit_{t'} = s.invokedCommit_{t'} = f(s).invokedCommit_{t'}$ for all $t' \in \mathcal{T}$.

Therefore, by the effect of $\mathsf{beginOk}_t$ in TMS1, we have $(f(s), \mathsf{beginOk}_t, f(s')) \in trans(\mathsf{TMS1})$. □

Lemma 31 If $(s, \mathsf{inv}_t(read(l)), s') \in trans(\mathsf{TMS2})$ then $(f(s), \mathsf{inv}_t(read(l)), f(s')) \in trans(\mathsf{TMS1})$.

Proof. $\mathsf{inv}_t(read(l))$ is enabled in $f(s)$ since by the definition of f, $f(s).status_t = s.pc_t = \mathsf{ready}$.
By the definition of f and the effect of $\mathsf{inv}_t(read(l))$ in TMS2, we have:

- $f(s').extOrder = s'.extOrder = s.extOrder = f(s).extOrder$.
- $f(s').status_t = abs(s'.pc_t) = abs(\mathsf{doRead}(l)) = \mathsf{opPending}$.
- $f(s').status_{t'} = abs(s'.pc_{t'}) = abs(s.pc_{t'}) = f(s).status_{t'}$ for all $t' \neq t$.
- $f(s').ops_{t'} = s'.ops_{t'} = s.ops_{t'} = f(s).ops_{t'}$ for all $t' \in \mathcal{T}$.
- $f(s').pendingOp_t = s'.pendingOp_t = read(l)$
- $f(s').pendingOp_{t'} = s'.pendingOp_{t'} = s.pendingOp_{t'} = f(s).pendingOp_{t'}$ for all $t' \neq t$.
- $f(s').invokedCommit_{t'} = s'.invokedCommit_{t'} = s.invokedCommit_{t'} = f(s).invokedCommit_{t'}$ for all $t' \in \mathcal{T}$.

Therefore, by the effect of $\mathsf{inv}_t(read(l))$ in TMS1, we have $(f(s), \mathsf{inv}_t(read(l)), f(s')) \in trans(\mathsf{TMS1})$. □

Lemma 32 If $(s, \mathsf{inv}_t(write(l, v)), s') \in trans(\mathsf{TMS2})$ then $(f(s), \mathsf{inv}_t(write(l, v)), f(s')) \in trans(\mathsf{TMS1})$.

Proof. $\mathsf{inv}_t(write(l,v))$ is enabled in $f(s)$ since by the definition of f, $f(s).status_t = s.pc_t = \mathsf{ready}$.
By the definition of f and the effect of $\mathsf{inv}_t(write(l,v))$ in TMS2, we have:

- $f(s').extOrder = s'.extOrder = s.extOrder = f(s).extOrder$.
- $f(s').status_t = abs(s'.pc_t) = abs(\mathsf{doWrite}(l,v)) = \mathsf{opPending}$.
- $f(s').status_{t'} = abs(s'.pc_{t'}) = abs(s.pc_{t'}) = f(s).status_{t'}$ for all $t' \neq t$.
- $f(s').ops_{t'} = s'.ops_{t'} = s.ops_{t'} = f(s).ops_{t'}$ for all $t' \in \mathcal{T}$.
- $f(s').pendingOp_t = s'.pendingOp_t = write(l,v)$.
- $f(s').pendingOp_{t'} = s'.pendingOp_{t'} = s.pendingOp_{t'} = f(s).pendingOp_{t'}$ for all $t' \neq t$.
- $f(s').invokedCommit_{t'} = s'.invokedCommit_{t'} = s.invokedCommit_{t'} = f(s).invokedCommit_{t'}$ for all $t' \in \mathcal{T}$.

Therefore, by the effect of $\mathsf{inv}_t(write(l,v))$ in TMS1, we have $(f(s), \mathsf{inv}_t(write(l,v)), f(s')) \in trans(\mathsf{TMS1})$. □

Lemma 33 If $(s, \mathsf{resp}_t(r), s') \in trans(\mathsf{TMS2})$ and s is reachable then $(f(s), \mathsf{resp}_t(r), f(s')) \in trans(\mathsf{TMS1})$.

Proof. By the preconditions of $\mathsf{resp}_t(v)$ and $\mathsf{resp}_t(\mathrm{ok})$, $s.pc_t = \mathsf{readResp}(v)$ for some v or $s.pc_t = \mathsf{writeRespOk}$. In either case, by the definition of f, $f(s).status_t = \mathsf{opPending}$. By Invariant 20, $s.validIdx(t,n)$ for some n. Let $S = \{t' \mid t' \text{ in } s.T_k \text{ for some } k \leq n\}$ and $\sigma = s.T_{0..n}$. By Invariant 19 and the definition of f, $\sigma \in ser(S, s.extOrder) = ser(S, f(s).extOrder)$. By Lemma 27, $S \subseteq f(s).VT$ and $f(s).extConsPrefix(S \cup \{t\})$. By Invariant 25 and the definitions of f and $doneOps_t$, $f(s).ops(\sigma \cdot t) \cdot (f(s).pendingOp_t, r) = s.ops(\sigma) \cdot s.ops_t \cdot (s.pendingOp_t, r) = s.ops(\sigma) \cdot s.doneOps_t$ is a legal sequential history. Thus, $f(s).validResp(t, f(s).pendingOp_t, r)$, so $\mathsf{resp}_t(r)$ is enabled in $f(s)$.
By the definition of f and the effects of $\mathsf{resp}_t(v)$ and $\mathsf{resp}_t(\mathrm{ok})$ in TMS2, we have:

- $f(s').extOrder = s'.extOrder = s.extOrder = f(s).extOrder$.
- $f(s').status_t = abs(s'.pc_t) = \mathsf{ready}$.
- $f(s).status_{t'} = abs(s.pc_{t'}) = abs(s.pc_{t'}) = f(s).status_{t'}$ for all $t' \neq t$.
- $f(s').ops_t = s'.ops_t = s.ops_t \cdot (s.pendingOp_t, r) = f(s).ops_t \cdot (f(s).pendingOp_t, r)$.
- $f(s').ops_{t'} = s'.ops_{t'} = s.ops_{t'} = f(s).ops_{t'}$ for all $t' \neq t$.
- $f(s').pendingOp_{t'} = s'.pendingOp_{t'} = s.pendingOp_{t'} = f(s).pendingOp_{t'}$ for all $t' \in \mathcal{T}$.
- $f(s').invokedCommit_{t'} = s'.invokedCommit_{t'} = s.invokedCommit_{t'} = f(s).invokedCommit_{t'}$ for all $t' \in \mathcal{T}$.

Therefore, by the effect of $\mathsf{resp}_t(r)$ in TMS1, we have $(f(s), \mathsf{resp}_t(r), f(s')) \in trans(\mathsf{TMS1})$. □

Lemma 34 If $(s, \mathsf{commit}_t, s') \in trans(\mathsf{TMS2})$ then $(f(s), \mathsf{commit}_t, f(s')) \in trans(\mathsf{TMS1})$.

Proof. commit_t is enabled in $f(s)$ since by the definition of f, $f(s).status_t = s.pc_t = \mathsf{ready}$.
By the definition of f and the effect of commit_t in TMS2, we have:

- $f(s').extOrder = s'.extOrder = s.extOrder = f(s).extOrder$.
- $f(s').status_t = abs(s'.pc_t) = abs(\mathsf{doCommit}) = \mathsf{commitPending}$.
- $f(s').status_{t'} = abs(s'.pc_{t'}) = abs(s.pc_{t'}) = f(s).status_{t'}$ for all $t' \neq t$.
- $f(s').ops_{t'} = s'.ops_{t'} = s.ops_{t'} = f(s).ops_{t'}$ for all $t' \in \mathcal{T}$.
- $f(s').pendingOp_{t'} = s'.pendingOp_{t'} = s.pendingOp_{t'} = f(s).pendingOp_{t'}$ for all $t' \in \mathcal{T}$.
- $f(s').invokedCommit_t = s'.invokedCommit_t = \mathbf{true}$.
- $f(s').invokedCommit_{t'} = s'.invokedCommit_{t'} = s.invokedCommit_{t'} = f(s).invokedCommit_{t'}$ for all $t' \neq t$.

Therefore, by the effect of commit_t in TMS1, we have $(f(s), \mathsf{commit}_t, f(s')) \in trans(\mathsf{TMS1})$. □

Lemma 35 If $(s, \mathsf{commitOk}_t, s') \in trans(\mathsf{TMS2})$ and s is reachable then $(f(s), \mathsf{commitOk}_t, f(s')) \in trans(\mathsf{TMS1})$.

Proof. By the precondition of $\mathsf{commitOk}_t$ and the definition of f, we have $f(s).status_t = abs(s.pc_t) = abs(\mathsf{commitRespOk}) = \mathsf{commitPending}$. By Lemma 26, $f(s).validCommit(t)$ (since $s.pc_t = \mathsf{commitRespOk}$), so $\mathsf{commitOk}_t$ is enabled in $f(s)$.
By the definition of f and the effects of $\mathsf{commitOk}_t$ in TMS2, we have:

- $f(s').extOrder = s'.extOrder = s.extOrder = f(s).extOrder$.

- $f(s').status_t = abs(s'.pc_t) = \mathsf{committed}$.
- $f(s).status_{t'} = abs(s.pc_{t'}) = abs(s.pc_{t'}) = f(s).status_{t'}$ for all $t' \neq t$.
- $f(s').ops_{t'} = s'.ops_{t'} = s.ops_{t'} = f(s).ops_{t'}$ for all $t' \in \mathcal{T}$.
- $f(s').pendingOp_{t'} = s'.pendingOp_{t'} = s.pendingOp_{t'} = f(s).pendingOp_{t'}$ for all $t' \in \mathcal{T}$.
- $f(s').invokedCommit_{t'} = s'.invokedCommit_{t'} = s.invokedCommit_{t'} = f(s).invokedCommit_{t'}$ for all $t' \in \mathcal{T}$.

Therefore, by the effect of $\mathsf{commitOk}_t$ in TMS1, we have $(f(s), \mathsf{commitOk}_t, f(s')) \in trans(\mathsf{TMS1})$. □

Lemma 36 If $(s, \mathsf{cancel}_t, s') \in trans(\mathsf{TMS2})$ then $(f(s), \mathsf{cancel}_t, f(s')) \in trans(\mathsf{TMS1})$.

Proof. cancel_t is enabled in $f(s)$ since by the definition of f, $f(s).status_t = s.pc_t = \mathsf{ready}$.

By the definition of f and the effect of cancel_t in TMS2, we have:

- $f(s').extOrder = s'.extOrder = s.extOrder = f(s).extOrder$.
- $f(s').status_t = abs(s'.pc_t) = \mathsf{cancelPending}$.
- $f(s').status_{t'} = abs(s'.pc_{t'}) = abs(s.pc_{t'}) = f(s).status_{t'}$ for all $t' \neq t$.
- $f(s').ops_{t'} = s'.ops_{t'} = s.ops_{t'} = f(s).ops_{t'}$ for all $t' \in \mathcal{T}$.
- $f(s').pendingOp_{t'} = s'.pendingOp_{t'} = s.pendingOp_{t'} = f(s).pendingOp_{t'}$ for all $t' \in \mathcal{T}$.
- $f(s').invokedCommit_{t'} = s'.invokedCommit_{t'} = s.invokedCommit_{t'} = f(s).invokedCommit_{t'}$ for all $t' \in \mathcal{T}$.

Therefore, by the effect of cancel_t in TMS1, we have $(f(s), \mathsf{cancel}_t, f(s')) \in trans(\mathsf{TMS1})$. □

Lemma 37 If $(s, \mathsf{abort}_t, s') \in trans(\mathsf{TMS2})$ and s is reachable then $(f(s), \mathsf{abort}_t, f(s')) \in trans(\mathsf{TMS1})$.

Proof. By the precondition of abort_t, $s.pc_t \notin \{\mathsf{notStarted}, \mathsf{ready}, \mathsf{commitRespOk}, \mathsf{committed}, \mathsf{aborted}\}$, so by the definition of f, $f(s).status_t = abs(s.pc_t) \in \{\mathsf{beginPending}, \mathsf{commitPending}, \mathsf{cancelPending}, \mathsf{opPending}\}$. By Lemma 26, $f(s).validFail(t)$ (since $s.pc_t \neq \mathsf{commitRespOk}$), so abort_t is enabled in $f(s)$.

By the definition of f and the effects of abort_t in TMS2, we have:

- $f(s').extOrder = s'.extOrder = s.extOrder = f(s).extOrder$.
- $f(s').status_t = abs(s'.pc_t) = \mathsf{aborted}$.
- $f(s).status_{t'} = abs(s.pc_{t'}) = abs(s.pc_{t'}) = f(s).status_{t'}$ for all $t' \neq t$.
- $f(s').ops_{t'} = s'.ops_{t'} = s.ops_{t'} = f(s).ops_{t'}$ for all $t' \in \mathcal{T}$.
- $f(s').pendingOp_{t'} = s'.pendingOp_{t'} = s.pendingOp_{t'} = f(s).pendingOp_{t'}$ for all $t' \in \mathcal{T}$.
- $f(s').invokedCommit_{t'} = s'.invokedCommit_{t'} = s.invokedCommit_{t'} = f(s).invokedCommit_{t'}$ for all $t' \in \mathcal{T}$.

Therefore, by the effect of abort_t in TMS1, we have $(f(s), \mathsf{abort}_t, f(s')) \in trans(\mathsf{TMS1})$. □

Finally, with these lemmas, it is straightforward to prove that TMS2 implements TMS1.

Theorem 1 TMS2 implements TMS1.

Proof. We prove that f is a refinement from TMS2 to TMS1, which implies that TMS2 implements TMS1 by Lemma 1. That f is a function follows immediately from its definition.

If s is a start state of TMS2 then

- $f(s).extOrder = s.extOrder$ is empty.
- $f(s).status_{t'} = abs(s.pc_{t'}) = \mathsf{notStarted}$ for all $t' \in \mathcal{T}$.
- $f(s).ops_{t'} = s.ops_{t'}$ is empty for all $t' \in \mathcal{T}$.
- $f(s).invokedCommit_{t'} = s.invokedCommit_{t'} = \mathbf{false}$ for all $t' \in \mathcal{T}$.

So $f(s)$ is a start state TMS1.

The step condition follows immediately from Lemmas 28 to 37. □

7. Related work

In an interesting exercise in 1987, participants in the Twenty-sixth Lake Arrowhead Workshop, "How Will We Specify Concurrent Systems in the Year 2000?", studied an informal specification of a serialisable database provided by Schneider [Sch92] and submitted formal specifications intended to capture the intent of the informal specification. Several approaches were submitted, critiqued, and refined, resulting in a special journal issue [Sch92] presenting the end results. The informal specification imposed no constraints on the behaviour of transactions that abort, and therefore neither do the resulting formalisations. As we have argued, at least for the context we have targetted, such constraints are necessary for TM implementations, and these constraints account for most of the complexity in our TMS1 condition.

We have given a precise semantics for TM that constrains transactions—even those that abort—to always observe consistent memory states. Our specification is defined using the I/O automaton model, which supports formal and machine-checkable verification of runtime libraries using well-established proof methodologies and tools, without limiting the structure of implementations or the number of transactions or variables they support. To our knowledge, none of the proposals discussed below achieves all of these goals.

The first published semantics for TM runtime libraries, due to Scott [Sco06], is most concerned with describing TM implementations in terms of various notions of conflicts, which dictate the circumstances under which transactions abort. The semantics imposes no constraints on operations of transactions that do abort. Furthermore, it is not expressed in a manner that lends itself to formal, machine-checked proofs of implementations.

Cohen et al. [COP+07, CPZ08] adapt Scott's semantics to a framework that facilitates such formal verification. They define automata-based models that are parameterised by a set of *admissible interchanges*. The set of admissible interchanges describes the operations that may be transposed when attempting to construct a legal sequential history from a given transactional history, and are chosen to capture Scott's various notions of conflict. Using their models, the authors are able to model check small instances of several simple transactional memory protocols that they developed as case studies. They also verified a simple protocol using the PVS theorem-proving system [PVS]. To our knowledge, this is the only other work that attempts to verify the correctness of a TM implementation at such a level of rigour. However, like Scott's conditions [Sco06], from which they derive, these automata-based models do not constrain the behaviour of aborted transactions. Cohen et al. also add *nontransactional* operations to their framework. We discuss nontransactional operations further in the next section.

Guerraoui et al. [GHJS08] define a version of opacity—called *abort consistency*—that is amenable to model checking. The authors prove that, for any implementation that satisfies certain *structural properties*, it suffices to prove that the implementation satisfies abort consistency with only two threads and two variables in order to prove that it satisfies abort consistency in general. This allows them to prove that such implementations satisfy abort consistency in general by using a model checker to prove it for these small instances. This work is extended in [GHS08] and [GHS09] to deal, respectively, with implementations that employ nondeterministic contention managers, and implementations defined over an underlying shared memory that satisfies only a relaxed consistency model.

Reducing the problem of verifying a general transactional memory implementation to verifying small instances that are amenable to model checking is an interesting approach. However, to date this approach does not meet our goal of achieving formal, machine-checked proofs showing that a TM system is correct. In particular, the conclusion that a TM system is correct depends on verifying that it satisfies the structural properties, and on the correctness of the result [GHJS08] showing that it suffices to prove smaller instances correct. To our knowledge, neither of these steps has been proved with the level of formal rigour needed to achieve machine-checked proofs. Furthermore, the approach is subject to the limitations of the model checker used, so that a small change to a TM implementation may result in the implementation no longer being amenable to verification using this approach.

Finally, there are important differences between the abort consistency property considered in these papers and opacity [GK08]. The read and write operations of [GHJS08] do not exhibit the values that are putatively read or written. A history is deemed opaque if there exists a sequential history that contains the same operations and that preserves the order of operations that *conflict*, according to a notion of conflict meant to capture opacity. This is in contrast to [GK08], where the sequential history must satisfy the specification of the underlying sequential object. To our knowledge, no formal or detailed account of the relationship between these two notions of correctness has been presented.

O'Leary et al. [OST09] have used Spin to verify small instances of the practical McRT software TM

implementation [SATH+06]. They use *serialisability* as the correctness condition, which requires that, for each execution, there exists a legal sequential history containing precisely the operations of the committed transactions in the execution, in which the operations of one transaction are not interleaved with those of another. This condition does not address the behaviour of transactions that abort, nor does it require the order of transactions to respect the external order, as TMS1 does. Initially, the authors used strict serialisability (see Section 1) as the correctness condition, which does require that the sequential order respect the external order. Interestingly, using their model-checking approach, the authors discovered that McRT does not meet this stronger specification, as had been expected. This demonstrates the significant value of model checking approaches, even when applied only to small instances.

We expect that TM runtime libraries will be most useful in the implementation of advanced language features that support shared-memory concurrency, such as atomic blocks or conditional critical regions [Hoa72]. Specifying semantics of such features and proving that an implementation provides the specified semantics is an important topic, which has been considered by some researchers [ABHI11, MG08]. Our work is at a different level, and is complementary to this work. For example, when these language features are implemented using a combination of compiler support and a TM runtime library, specifying and proving properties of the runtime library supports modular reasoning about the correctness of the implemented language features, as well as the ability to "plug in" different runtime libraries that have been proved to implement the same correctness condition. Therefore, we believe that notions of TM correctness like TMS1, as well as verifications of TM implementations, can contribute significantly to the development of reliable language features for shared-memory concurrency, as well as programs that use TM runtime libraries directly.

8. Ongoing and future work

We have introduced a general TM correctness condition TMS1, and a more restrictive but more intuitive condition TMS2. We have completed a formal, machine-checked proof that TMS2 implements TMS1, guided by the manual proof presented in the previous section.

In ongoing work, we aim to prove that some popular and practical TM implementations implement TMS2. Indeed, we have recently formalised and verified the NOrec algorithm [DSS10], a simple yet practical TM algorithm. To our knowledge, this is the first fully formal machine-checked proof of a practical TM algorithm (i.e., one that was designed with practical goals rather than as a case study for verification). To verify NOrec, we followed the approach outlined in the introduction, first formalising a version of NOrec that uses coarser-grained synchronisation than is consistent with current multiprocessor architectures and proving that it implements TMS2, and then refining this version to successively more realistic implementations, ultimately arriving at a version that straightforwardly encodes the pseudocode in the paper as an IOA. (We did this encoding by hand, but one can imagine a tool that would do so automatically.) Although many of the details in the proof are, of course, specific to NOrec, we believe that this proof is a useful guide to future verification of other TM implementations.

Although we verified NOrec using only forward simulations, some TM implementations, particularly nonblocking ones, require *backward simulations* [LV95], which are more challenging to prove. To facilitate the verification of such implementations, as in our previous work [CGLM06, DM09], we plan to identify one or more general *intermediate automata* that we can prove (using backward simulation proofs) implement TMS2. The idea is that we and others can then prove the correctness of a TM implementation by proving that it implements one of these intermediate automata using only forward simulation.

There are numerous aspects of TM models and implementations that our work does not yet address. First, for this work we have targetted runtime libraries for unmanaged languages, in which the ability to avoid transactions observing inconsistent behaviour is critical because, in general, this could cause unrecoverable errors such as divide-by-zero, segmentation violations, overwriting code, etc. This requirement heavily influences the choice of correctness condition. In contrast, managed runtimes must prevent such unrecoverable errors regardless of what code is executed, and therefore it is reasonable to consider more relaxed correctness conditions that allow transactions to observe inconsistent behaviour temporarily, and detect and recover from it. Specifying correctness conditions for such contexts is of interest.

Even in unmanaged contexts, with adequate compiler support, it is acceptable for transactions to observe inconsistent behaviour while executing "nondangerous" code (which is proved by the compiler not to cause unrecoverable errors even if it observes inconsistent behaviour), but the transaction must again be known to have observed consistent behaviour before executing "dangerous" code that is not, or cannot be proved to be,

safe to execute with inconsistent data. For example, the compiler-library interface for the SkySTM library [LLM+09, Sky09] is designed to allow optimisations that avoid validation (checking whether the transaction is consistent) if the subsequent code the transaction will execute is nondangerous.

Similarly, compiler analysis may in some cases be able to provide useful information to the TM runtime about the future behaviour of a transaction. For example, if the TM runtime were informed that a transaction had performed its last write and that it would eventually attempt to commit, then we could relax the requirement that transactions observe effects only of transactions that have already invoked commit. To model the use of such information, we could extend TMS1's interface to allow the information to be passed to the runtime, and modify the correctness condition appropriately to allow it to exploit the information. Because our basic interface does not facilitate such information, our correctness condition necessarily forbids transactions from observing any effects of a transaction that has not yet invoked commit.

Our work to date on specifying and verifying TM systems that support read-write memory assumes that locations are accessed only via read and write operations within transactions. It is also interesting to consider issues that arise when memory locations are accessed *nontransactionally*. Previous work aimed at verification of TM [CPZ08] considers nontransactional accesses in the sense of accesses that must not abort and should be atomic with respect to other nontransactional accesses and to transactions. It would be straightforward to extend our correctness conditions to take such accesses into account.

However, in our opinion, it is more important and more challenging to consider nontransactional memory accesses in the sense usually used in the TM literature, namely *uninstrumented* load and store instructions accessing memory locations that are also accessed within transactions. Among other reasons, this is important to allow TM to be used in conjunction with code for which recompilation or binary translation is not practical. This may be the case for performance, technical, business, or legal reasons.

At least in conventional architectures without special hardware support for TM, it is impossible to make guarantees as strong as those considered in [CPZ08] if uninstrumented nontransactional loads and stores can access locations concurrently with transactions accessing the same locations. Therefore, such accesses are usually forbidden. For example, in the Draft Specification of Transactional Language Constructs for C++ [ATe09], the semantics of a program that performs such accesses are not defined. Even so, the precise definition of "concurrently" is important, and formalising the assumption that nontransactional accesses are not performed concurrently with transactional accesses, as well as specifying what it means for nontransactional accesses to be correctly synchronised and proving that they behave as specified in a given implementation would be useful.

Our work to date does not consider various other important aspects of TM systems, including various forms of nesting [MH06] and other extensions such as boosting [HK08], nor does it consider progress properties. All of these are interesting possible directions for future work.

9. Concluding remarks

We have introduced a new correctness condition TMS1 that is intended to specify correct behaviour of transactional memory runtime libraries for unmanaged environments. A key aspect of correctness in such environments is the requirement that transactions never observe inconsistent behaviour, even if they will ultimately abort.

We have defined TMS1 using an I/O automaton to make the definition precise, and to facilitate the construction of precise, machine-checked proofs that TM systems (also modelled using I/O automata) are correct. To further facilitate such proofs, we have also presented a more restrictive condition TMS2 and proved that any implementation that satisfies TMS2 also satisfies TMS1. TMS2 is closer in structure and intuition to a variety of practical TM implementations, so this "hierarchical" approach makes it easier to prove a TM implementation correct.

While our work lays an important foundation in specifying correct behaviour of a TM runtime system, there is plenty left to do, both in developing similar specifications for other contexts or to support more features, and in completing machine-checked proofs for a variety of TM implementations.

Acknowledgments We thank the anonymous reviewers whose careful reading and insightful comments led us to significantly expand and improve the exposition, and even to simplify the proof by eliminating an unnecessary invariant.

References

[AA08] Utku Aydonat and Tarek Abdelrahmen. Serializability of transactions in software transactional memory. In *3rd ACM Workshop on Transactional Computing (TRANSACT)*, 2008.

[ABHI11] Martín Abadi, Andrew Birrell, Tim Harris, and Michael Isard. Semantics of transactional memory and automatic mutual exclusion. *ACM Trans. Program. Lang. Syst.*, 33(1):2:1–2:50, January 2011.

[ATe09] Ali-Reza Adl-Tabatabai and Tatiana Shpeisman (eds). Draft specification of transactional language constructs for C++, Version 1.0. http://labs.oracle.com/scalable/pubs/C++-transactional-constructs-1.0.pdf, August 2009.

[CDG05] Robert Colvin, Simon Doherty, and Lindsay Groves. Verifying concurrent data structures by simulation. In Eerke Boiten and John Derrick, editors, *Proceedings of the RefineNet Workshop (REFINE), Guildford, UK*, Electronic Notes in Theoretical Computer Science, 2005.

[CGLM06] Robert Colvin, Lindsay Groves, Victor Luchangco, and Mark Moir. Formal verification of a lazy concurrent list-based set algorithm. In *Proceedings of the 18th International Conference on Computer Aided Verification (CAV)*, pages 475–488, 2006.

[COP+07] Ariel Cohen, John W. O'Leary, Amir Pnueli, Mark R. Tuttle, and Lenore D. Zuck. Verifying correctness of transactional memories. In *Proceedings of the Formal Methods in Computer Aided Design (FMCAD)*, pages 37–44, 2007.

[CPZ08] Ariel Cohen, Amir Pnueli, and Lenore D. Zuck. Mechanical verification of transactional memories with non-transactional memory accesses. In *Proceedings of the 20th International Conference on Computer Aided Verification (CAV)*, pages 121–134, 2008.

[DGLM04] Simon Doherty, Lindsay Groves, Victor Luchangco, and Mark Moir. Formal verification of a practical lock-free queue algorithm. In *Proceedings of the International Conference on Formal Techniques for Networked and Distributed Systems (FORTE)*, pages 97–114, 2004.

[DGLM09] Simon Doherty, Lindsay Groves, Victor Luchangco, and Mark Moir. Towards formally specifying and verifying transactional memory. In *Proceedings of the RefineNet Workshop (REFINE)*, Electronic Notes in Theoretical Computer Science, 2009. Available from http://labs.oracle.com/scalable/pubs/Refine09-TM-correctness.pdf.

[DM09] Simon Doherty and Mark Moir. Nonblocking algorithms and backward simulation. In *Proceedings of 23rd International Conference on Distributed Computing (DISC)*, 2009.

[DSS06] Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking II. In *Proceedings of the International Conference on Distributed Computing (DISC)*, pages 194–208, 2006.

[DSS10] Luke Dalessandro, Michael F. Spear, and Michael L. Scott. NOrec: Streamlining STM by abolishing ownership records. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 67–78, 2010.

[GHJS08] Rachid Guerraoui, Thomas A. Henzinger, Barbara Jobstmann, and Vasu Singh. Model checking transactional memories. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 372–382, 2008.

[GHS08] Rachid Guerraoui, Thomas A. Henzinger, and Vasu Singh. Completeness and nondeterminism in model checking transactional memories. In *Proceedings of the 19th International Conference on Concurrency Theory (CONCUR)*, pages 21–35, 2008.

[GHS09] Rachid Guerraoui, Thomas A. Henzinger, and Vasu Singh. Software transactional memory on relaxed memory models. In *Proceedings of the 21st International Conference on Computer Aided Verification (CAV)*, pages 321–336, 2009.

[GK08] Rachid Guerraoui and Michal Kapalka. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 175–184, 2008.

[GK10] Rachid Guerraoui and Michał Kapałka. *Principles of Transactional Memory*. Synthesis Lectures on Distributed Computing Theory. Morgan Claypool, 2010.

[HK08] Maurice Herlihy and Eric Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 207–216, 2008.

[HM93] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA)*, 1993.

[Hoa72] C. A. R. Hoare. Towards a theory of parallel programming. In *Operating Systems Techniques*, pages 61–71. Academic Press, 1972.

[HSATH06] Richard L. Hudson, Bratin Saha, Ali-Reza Adl-Tabatabai, and Benjamin C. Hertzberg. McRT-Malloc: a scalable transactional memory allocator. In *Proceedings of the 5th International Symposium on Memory Management (ISMM)*, pages 74–83, 2006.

[IdMR08] Damien Imbs, José de Mendívil, and Michel Raynal. On the consistency conditions of transactional memories. Technical Report 1917, Institut de Recherche en Informatique et Systèmes Aalátoires, 2008.

[IdMR09] Damien Imbs, José de Mendívil, and Michel Raynal. Brief announcement: Virtual world consistency, a new condition for STM systems. In *Proceedings of the 2009 ACM Symposium on Principles of Distributed Computing (PODC)*, pages 280–281, 2009.

[LLM+09] Yossi Lev, Victor Luchangco, Virendra J. Marathe, Mark Moir, Dan Nussbaum, and Marek Olszewski. Anatomy of a Scalable Software Transactional Memory. In *4th ACM SIGPLAN Workshop on Transactional Computing (TRANSACT)*, 2009.

[LT87] N. Lynch and M. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 137–151, 1987.

[LT89] Nancy A. Lynch and Mark R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2:219–246, 1989.

[LV95] Nancy Lynch and Frits Vaandrager. Forward and backward simulations, I: Untimed systems. *Information and Computation*, 121(2):214–233, September 1995.

[MG08] Katherine F. Moore and Dan Grossman. High-level small step operational semantics for transactions. In *Proceedings of the 35th Annual ACM Symposium on Principles of Prgoramming Languages (POPL)*, 2008.

[MH06] J. Eliot B. Moss and Antony L. Hosking. Nested transactional memory: model and architecture sketches. *Sci. Comput. Program.*, 63(2):186–201, 2006.

[OG76] Susan Owicki and David Gries. An axiomatic proof technique for parallel programs. *Acta informatica*, 6(4):319–340, 1976.

[OST09] John O'Leary, Bratin Saha, and Mark R. Tuttle. Model checking transactional memory with Spin. In *Proceedings of the 29th IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 335–342, 2009.

[Pap79] Christos H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26:631–653, October 1979.

[PVS] The PVS Specification and Verification System, http://pvs.csl.sri.com/.

[RRW08] Hany E. Ramadan, Indrajit Roy, and Emmett Witchel. Dependence-Aware Transactional Memory for Increased Concurrency. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture*, pages 246–257, 2008.

[SATH+06] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In *Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 187–197, 2006.

[Sch92] Fred Schneider. Introduction, special issue: Specification of concurrent systems. *Distrib. Comput.*, 6(1), 1992.

[Sco06] Michael L. Scott. Sequential specification of transactional memory semantics. In *1st ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT)*, 2006.

[Sky09] SkySTM Interest Google Group. http://groups.google.com/group/skystm-interest, 2009.