Verifying Correctness of Transactional Memories

Ariel Cohen (CS/CIMS/NYU, arielc@cs.nyu.edu) John W. O'Leary (Intel, john.w.oleary@intel.com) Amir Pnueli (CS/CIMS/NYU, amir@cs.nyu.edu) Mark R. Tuttle (Intel, tuttle@acm.org) Lenore D. Zuck (CS/UIC, lenore@cs.uic.edu)

***Abstract*—We show how to verify the correctness of transactional memory implementations with a model checker. We show how to specify transactional memory in terms of the admissible interchange of transaction operations, and give proof rules for showing that an implementation satisfies this specification. This notion of an admissible interchange is a key to our ability to use a model checker, and lets us capture the various notions of transaction conflict as characterized by Scott. We demonstrate our work using the TLC model checker to verify several well-known implementations described abstractly in the TLA$^+$ specification language.**

***Index Terms*—Verification, transactional memory, model checking, HTM, STM, TLA$^+$, TLC.**

I. Introduction

The most important development in processor architecture in the last decade has been the shift from single-threaded, single-core processors to multi-threaded, multi-core processors. Taking advantage of these new processors, however, requires rewriting our applications as multi-threaded programs, and multi-threaded programs are hard to write, especially when several threads need to access the same data. Conventional approaches employ locks to regulate access to shared data, but locks are subtle and hard to use correctly. Some well-known problems with locks are *priority inversion*, which can occur when a low priority thread holds a lock needed by a higher priority thread; and *deadlock*, which can occur when several threads attempt to acquire the same set of locks in a different order.

Transactional memory [1] is a programming abstraction intended to simplify the synchronization of conflicting memory accesses (by concurrent threads) without the headaches associated with locks. A *transaction* is a sequence of memory operations that appears to be performed atomically with respect to other memory operations. The idea is that if a concurrent program is written so that each access to a shared data structure is encapsulated within a transaction, then all reads and writes to the data structure will appear to occur in isolation in some sequential order, and the established theory of database serializability will help us reason about the correctness of such programs. Early hardware implementations of transactional memory were limited to relatively small transactions, but recent software implementations (sometimes depending on limited hardware support) have managed to remove this restriction.

Larus and Rajwar [2] survey nearly 40 implementations of transactional memory in their comprehensive book on the subject, which differ in many dimensions. An implementation may employ *eager version control* (or *direct update*) in which a transaction modifies an object in place and restores the object to its original value upon abort, or may employ *lazy version control* (or *deferred update*) in which a transaction modifies a private copy of the object and overwrites the object with this private copy upon commit. An implementation may support *weak atomicity* or *strong atomicity* depending on whether the implementation guarantees transactional semantics only to object references within transactions or to all object references (even those outside of transactions). Different implementations may use very different approaches to detecting conflicts among transactions such as *lazy or eager invalidation*, and support many different progress conditions in the presence of contention such as *wait freedom*, *lock freedom*, or *obstruction freedom*.

Scott [3] wrote a widely-cited paper that was the first to characterize transactional memory in a way that captured and clarified the many semantic distinctions among the most popular implementations. His approach was to begin with classical notions of transactional histories and sequential specifications, and to introduce two important notions. The first was a *conflict function* which specifies when two transactions cannot both succeed (a safety condition). The second was an *arbitration function* which specifies which of two transactions must fail (a liveness condition). Scott's work went a long way toward making sense of transactional memory semantics, but his work was purely semantic and did not immediately facilitate mechanical verification of implementations.

In this paper, we present an abstract model for specifying transactional memory semantics inspired by Scott's original work, and a proof rule for verifying that an implementation satisfies a transactional memory specification. The premisses of our proof rule can be checked with a model checker, and we demonstrate the method by modeling three well-known transactional memory implementations in TLA$^+$ and proving their correctness with the model checker TLC. The essential contribution of this paper that enables mechanical checking is the notion of an *admissible interchange* used to model the approaches to conflict detection and resolution characterized by Scott in his paper. The work we report here is preliminary, but we hope it will form the basis for analysis of well-known issues like *privatization* and *granular lost update* in addition to implementation correctness, and for analysis of the interaction between hardware and software support for transactional memory.

The rest of this paper is organized as follows. We begin with preliminary definitions related to transactions and transaction sequences in Section II, and we define an admissible interchange in Section III. This definition is the key to our ability to model check transactional memory implementations, and we show how Scott's transaction conflict classes can be characterized in terms of admissible interchanges. We give our specification of a transactional memory and what it means for an implementation to be correct in Section IV, and we give proof rules for verifying implementation correctness in Section V. We sketch the correctness proofs for several implementations of transactional memory in Section VI, and show how to use a model checker to verify their correctness in Section VII. Finally, in Section VIII, we end with some conclusions and open problems.

II. Transactional Sequences

Assume n *clients* that direct transactional requests to a *memory system*, denoted by *memory*. The requests that can be issued by client i are:

- $◀_i$ – An open transaction request.
- $R_i(x)$ – A read request from address $x \in \mathbb{N}$.
- $W_i(y, v)$ – A request to write the value $v \in \mathbb{N}$ to address $y \in \mathbb{N}$.
- $▶_i$ – A close transaction request.

The memory provides a response for each request. For requests that are rejected (e.g., a $◀_i$ request while client i has a pending

This research was supported in part by SRC grant 2004-TJ-1256 and ONR grant N00014-99-1-0131.

transaction) the memory returns an error flag. For requests that are accepted, and do not require a special response (e.g., $\blacktriangleleft_i$ when there is no pending i transaction), the memory responds with some acknowledgment. For accepted requests that require a response the memory provides a return value. For $R_i(x)$, it is a natural number indicating the value of the memory at location x. For $\blacktriangleright_i$, the memory responds with "commit" or "abort," according to its decision on whether the transaction should be *committed* or *aborted*.

Let $E_i\colon \{\blacktriangleleft_i, R_i(x,u), W_i(x,v), \blacktriangleright_i, \not\blacktriangleright_i\}$ be the set of *observable events* associated with client i, where $\not\blacktriangleright_i$ represents the closing of a transaction that has been aborted (while $\blacktriangleright_i$ represents the closing of a transaction that has been committed). We consider as observable only requests that are accepted, and we include the memory's response for $R_i(x)$ and $\blacktriangleright_i$ requests (rather than the requests themselves). In this paper we also mandate that the order in which the memory issues its commit responses (and therefore the order of observable $\blacktriangleright_i$ events) uniquely determines the order of committed transactions. Let E be the set of all observable events over all clients, i.e., $E = \bigcup_{i=1}^{n} E_i$.

Note that we have defined $R_i(x)$ to be the request corresponding to the response $R_i(x,u)$, and that we are abusing notation by writing $\blacktriangleleft_i$, $W_i(y,v)$, $\blacktriangleright_i$ to denote both a request and a corresponding response when the meaning is clear from context. We will also denote responses $R_i(x,u)$ and $W_i(x,v)$ by R_i and W_i when the exact values of the parameters are unimportant or are clear from context.

Let $\sigma\colon e_0, e_1, \dots, e_k$ be a finite sequence of observable E-events. The sequence σ is called a *well-formed transactional sequence* (TS for short) if the following conditions hold:

1) For every client i, let $\sigma|_i$ be the sequence obtained by projecting σ onto E_i. Then $\sigma|_i$ satisfies the regular expression T_i^*, where T_i is the regular expression $\blacktriangleleft_i\ (R_i{+}W_i)^*(\blacktriangleright_i + \not\blacktriangleright_i)$. For each occurrence of T_i in $\sigma|_i$, we refer to the first and last elements as *matching*. The notion of matching is lifted to σ itself, where $\blacktriangleleft_i$ and $\blacktriangleright_i$ (or $\not\blacktriangleright_i$) are matching if they are matching in $\sigma|_i$;
2) The sequence σ is *locally read-write consistent*: i.e, for any subsequence $W_i(x,v)\eta R_i(x,u)$ in σ, where η contains no event of the form $\blacktriangleright_i$, $\not\blacktriangleright_i$, or $W_i(x,w)$, we have $u = v$.

We denote by $\mathcal{T}$ the set of all well-formed transactional sequences, and by *pref*$(\mathcal{T})$ the set of prefixes of such sequences.

Notice that the requirement of local read-write consistency can be enforced by each client locally. To build on this observation, we assume that, within a single transaction, there is no $R_i(x)$ following a $W_i(x)$, and there are no two reads or two writes to the same address. As a result, we can assume that the sequence of events constituting a single i-transaction has the form

$$\blacktriangleleft_i\ R_i(x_1,u_1)\cdots R_i(x_r,u_r)W_i(y_1,v_1)\cdots W_i(y_w,v_w)\{\blacktriangleright_i, \not\blacktriangleright_i\}$$

where the addresses in each of the sequences $x_1,\dots,x_r$ and $y_1,\dots,y_w$ are pairwise distinct. With this assumption, the requirement of local read-write consistency is always (vacuously) satisfied.

The TS σ is called *atomic* if:

1) It satisfies the regular expression $(T_1 + \cdots + T_n)^*$. That is, there is no overlap between any two transactions.
2) The sequence σ is *globally read-write consistent*: for any subsequence $W_i(x,v)\eta R_j(x,u)$ in σ, where η contains $\blacktriangleright_i$ but contains no event $W_k(x,\cdot)$ followed by an event $\blacktriangleright_k$, it is the case that $u = v$.

III. Interchanging Events

When is a TS σ a correct behavior of a transactional memory implementation? It is natural to say that σ is correct if it can be transformed into an atomic TS by first removing from it all events that belong to aborted transactions, then freely interchanging adjacent events that belong to committed transactions. This correctness criterion is known as *serializability*. Since we require that the order of $\blacktriangleright_i$ events determines the order of committed transactions, we choose to disallow the interchange of $\blacktriangleright$ events. This narrower criterion is known as *strict serializability*, and we will further refine it throughout the rest of this section.

Strict serializability, by itself, is far from a satisfactory correctness criterion for TM implementations. Let us say that transactions T_i and T_j overlap when $\blacktriangleleft_i$ precedes $\blacktriangleright_j$ and $\blacktriangleleft_j$ precedes $\blacktriangleright_i$, and suppose we wish to specify a class of implementations that forbid two overlapping transactions to both commit. Strict serializability is much too generous a specification, as many strictly serializable transactional sequences contain overlapping transactions. Scott [3] introduced *conflicts* to describe the TS's characteristic of different classes of implementations (in Scott's terminology, our hypothetical class of implementations avoids *overlap conflicts*). We will describe conflicts by restricting which events can be exchanged during serialization. To specify the class of implementations that forbid overlapping transactions, for example, we will add the restriction that adjacent $\blacktriangleleft$ and $\blacktriangleright$ events cannot be interchanged during serialization: thus no TS with overlapping events will be strictly serializable.

Before introducing our notion of admissible interchanges, we briefly describe Scott's six classes of conflicts. For a TS σ, let $\prec_\sigma$ denote the precedence relation of events in σ, meaning that $e_i \prec_\sigma e_j$ if e_i occurs before e_j in σ. We omit the σ subscript when its identity is clear from the context.

1) A TS σ has an *overlap conflict* if for some transactions T_i and T_j, we have $\blacktriangleleft_i \prec \blacktriangleright_j$ and $\blacktriangleleft_j \prec \blacktriangleright_i$.
2) A TS σ has a *writer overlap conflict* if two transactions overlap and one performs a write before the other terminates, i.e., for some T_i and T_j, we have $\blacktriangleleft_i \prec W_j \prec \blacktriangleright_i$ or $W_j \prec \blacktriangleleft_i \prec \blacktriangleright_j$.
3) A TS has a *lazy invalidation* conflict if commitment of one transaction may invalidate a read of the other, i.e., if for some transaction T_i and T_j and some memory address x, we have $R_i(x), W_j(x) \prec \blacktriangleright_j \prec \blacktriangleright_i$.
4) A TS has an *eager W-R* conflict if it has a lazy invalidation conflict, or if for some transactions T_i and T_j and some memory address x, we have $W_i(x) \prec R_j(x) \prec \blacktriangleright_i$.
5) A TS has a *mixed invalidation* conflict if it has a lazy invalidation conflict, or if for some transaction T_i and T_j, and some memory address x, we have $R_i(x) \prec W_i(x), W_j(x) \prec \blacktriangleright_i, \blacktriangleright_j$.
6) A TS has an *eager invalidation* conflict if it has an eager W-R conflict, or if for some transaction T_i and T_j and some memory address x, we have $R_i(x) \prec W_j(x) \prec \blacktriangleright_i$.

Let c be some conflict (e.g., "write overlap"). We denote by $\mathcal{F}_c$ the *resolving predicate* describing the interchanges that may resolve a c-conflict. For a pair of events $\langle e_i, e_j \rangle$ that belong to transactions T_i and T_j (where $i \neq j$), we denote by $\langle e_i, e_j \rangle \models \mathcal{F}_c$ the fact that $\mathcal{F}_c$ implies that the interchange $\langle e_i, e_j \rangle$ may resolve a c-conflict. In Fig. 1 we define $\models \mathcal{F}_c$ for each of Scott's conflicts c and every pair $\langle e_i, e_j \rangle$. In the full version of this paper we describe the language used to define the $\mathcal{F}$'s.

Given a conflict c and the resolving predicate $\mathcal{F}_c$ that corresponds to it, a TS σ is said to be *serializable* with respect to $\mathcal{F}_c$ if it can be transformed into an atomic TS by a sequence of admissible interchanges (that do not satisfy $\mathcal{F}_c$). Note that this definition is not equivalent to Scott's definition of c which, in some cases, may imply interchanges that are not admissible (that is, that satisfy $\mathcal{F}_c$).

The sequence $\widetilde{\sigma}$ is called the *purified version* of TS σ if $\widetilde{\sigma}$ is obtained by removing from σ all aborted transactions, i.e., removing the opening and closing events for such a transaction and all the read-write events by the same client that occurred between the opening and

Conflict (c)	$\langle e_i, e_j \rangle \models \mathcal{F}_c$ if:
Overlap (o)	$e_i = \blacktriangleleft_i \wedge e_j = \blacktriangleright_j$
Writer Overlap (*wo*)	$\exists x, u.(e_i = \blacktriangleleft_i \wedge e_j = W_j(x,u) \ \vee \ e_i = W_i(x,u) \wedge e_j \in \{\blacktriangleleft_j, \blacktriangleright_j\})$
Lazy Invalidation (*li*)	$\exists x, u, v.(W_j(x,u) \in T_j \wedge e_i = R_i(x,v) \wedge e_j = \blacktriangleright_j)$
Eager W-R (*ewr*)	$\mathcal{F}_{li} \ \vee \ (\exists x, u, v. e_i = W_i(x,u) \wedge e_j = R_j(x,v))$
Mixed Invalidation (*mi*)	$\mathcal{F}_{li} \ \vee \ \exists x, u, v.(e_i = R_i(x) \wedge e_j = W_j(x) \wedge e_i \prec W_i(x,u) \prec \blacktriangleright_j \wedge e_j \prec \blacktriangleright_i \ \vee$ $e_i = W_j(x,u) \wedge e_j = \blacktriangleright_j \wedge R_i(x,u) \prec W_i(x))$
Eager Invalidation (*ei*)	$\mathcal{F}_{ewr} \ \vee \ \exists x, u, v.(e_i = R_i(x,u) \wedge e_j = W_j(x,v) \wedge e_j \prec \blacktriangleright_i \ \vee \ e_i = W_i(x,u) \wedge e_j = \blacktriangleright_j \wedge R_j(x,v) \prec e_i)$

Fig. 1. Conflicts and Their Corresponding Predicates

closing events. When we specify the correctness of a transactional memory implementation, only the purified versions of the implementation's transaction sequences will have to be serializable.

IV. TM: Specification and Implementation

Let $\mathcal{F}$ be a resolving predicate which we fix for the remainder of this section. We now describe $Spec_{\mathcal{F}}$ – a specification of transactional memory that generates all TSs serializable with respect to $\mathcal{F}$ and a definition of a correct implementation of $Spec_{\mathcal{F}}$.

The specification $Spec_{\mathcal{F}}$ can be formally presented as an FDS (fair transition system, see Appendix). It uses the following data structures:

- *spec_mem*: $\mathbb{N} \mapsto \mathbb{N}$ — A persistent memory, represented as an array of naturals. For simplicity, we represent it as an infinite array. Initially, for every $i \geq 0$, *spec_mem*$[i] = 0$;
- q: **queue of** $E \cup \bigcup_{i=1}^{n} \{mark_i\}$ — A queue of pending events, initially empty;
- *spec_out*: scalar in $E_\perp = E \cup \{\perp\}$ — an output variable recording responses to clients, initially $\perp$;
- $doomed$: **array** $[1..n]$ **of booleans** — An array recording which transactions are doomed to be aborted. Initially $doomed[i] = \text{F}$ for every i.

Let

$$tr: \ \blacktriangleleft_i \ R_i(x_1,u_1), \ldots, R_i(x_r,u_r), W_i(y_1,v_1), \ldots, W_i(y_w,v_w) \ \blacktriangleright_i$$

be a transaction. We say that *tr* is *consistent* with *spec_mem* if, for each $j \in [1..r]$, *spec_mem*$[x_j] = u_j$. The *update* of *spec_mem* by *tr* is defined to be the memory *spec_mem'* such that, for each $j \in [1..w]$, *spec_mem'*$[y_j] = v_j$ and, for all $k \notin \{y_1, \ldots, y_w\}$, *spec_mem'*$[k] =$ *spec_mem*$[k]$.

Intuitively, the stream of *spec_out*'s is the sequence of observable events. Pending transactions are partitioned to two categories. *Active* transactions, whose events are maintained in q in the order they are in *spec_out*, and *doomed* transactions, that must be aborted, indicated by $doomed[i] = \text{T}$. When a transaction is doomed, all its events are removed from q, and subsequent events are echoes by *spec_out* but nowhere stored. When a pending transaction is committed, aborted, or doomed, all its events (which may be none if the transaction is doomed) are removed from q, and subsequent events are stored nowhere and it is marked as "undoomed." A transaction T_i is *doomed* if $doomed[i] = \text{T}$; T_i is *active* if q has some E_i-event; T_i is *inactive* if its neither active nor doomed.

For every active transaction T_i, we allow the queue q to include a special symbol, $mark_i$. The symbol $mark_i$ is added to the queue when a T_i issues a *c*lose request, and some tests are done to determine whether it can safely close. If the test is successful, *spec_out* is set to $\blacktriangleright_i$, otherwise, it is set to $\not\blacktriangleright_i$, and then $mark_i$ as well as all the E_i-events are removed from the queue. We say that q is marked (unmarked) if it has some (no) $mark_i$ symbol.

Transaction a_1–a_5 are applicable only when q is unmarked. Note that a_4 and a_5 do not set *spec_out* to a value. For such cases we assume that *spec_out* is set to $\perp$.

a_1. For some $i \in [1..n]$, if T_i is inactive, write $\blacktriangleleft_i$ to *spec_out* and append it to the end of the queue q.

a_2. For some $i \in [1..n]$, and $x, u \in \mathbb{N}$, if T_i is active or doomed, write $W_i(x,u)$ to *spec_out*. If T_i is active, then $W_i(x,u)$ is appended to the end of the queue q.

a_3. For some $i \in [1..n]$, and $x, u \in \mathbb{N}$, if T_i is active or doomed, write $R_i(x,u)$ to *spec_out*. If T_i is active, then $R_i(x,u)$ is appended to q. Moreover, in this case we also require that the events of T_i are locally consistent.

a_4. For some $i \in [1..n]$ such that T_i is active, remove all of events in E_i from the queue q and set $doomed[i]$ to T.

a_5 For some $i \in [1..n]$ such that T_i is active, add $mark_i$ to the end of q.

Transition a_6–a_8 deal with commits and aborts. It is a_7 that determines whether a transaction marked for commit can indeed commit.

a_6. For some $i \in [1..n]$ such that T_i is active or doomed, write $\not\blacktriangleright_i$ to *spec_out*, and remove all of E_i- and $mark_i$-events from the queue q, and set $doomed[i]$ to F.

a_7. For some $i \in [1..n]$ such that T_i is active, if T_i is consistent with *spec_out*, all of its events appear consecutively in the front of q, and $mark_i$ is in q, then write $\blacktriangleright_i$ to *spec_out*, update *spec_mem* according to T_i, and remove all E_i- and $mark_i$-events from the queue.

a_8. Interchange the order of two contiguous events e_i, e_j in q belonging to different transactions T_i and T_j, respectively, if $mark_j$ is in q, and $\langle e_i, e_j \rangle \not\models \mathcal{F}$. We treat $mark_j$ as if it is a $\blacktriangleright_j$ and assume a hypothetical $\blacktriangleright_i$ appended at the end of q.

a_9. An idling transition which does not modify *spec_mem*, q or $doomed$.

Note that the updates of the queue in a_4, a_6, and a_7 are not standard queue operations.

The specification has n associated justice requirements, namely, for every $i = 1, \ldots, n$:

there are infinitely many states in which $q|_i$ is empty.

A sequence σ over E^* is *compatible with* Spec$_{\mathcal{F}}$ if σ can be obtained by the sequence of *spec_out* which $Spec_{\mathcal{F}}$ outputs, once all the $\perp$'s are removed. We then have:

Claim 1: For every sequence σ over E, σ is compatible with $Spec_{\mathcal{F}}$ iff σ is serializable with respect to $\mathcal{F}$.

An *implementation TM*: (*read*, *close*) of a transactional memory consists of a pair of functions

$$\begin{array}{lcl} read & : & pref(\mathcal{T}) \times [1..n] \times \mathbb{N} \to \mathbb{N} \quad \text{and} \\ close & : & pref(\mathcal{T}) \times [1..n] \to \{commit, abort\} \end{array}$$

For a prefix σ of a TS, $read(\sigma, i, x)$ is the response (value) of the memory to an accepted $R_i(x)$ request immediately following σ, and $close(\sigma, i)$ is the response (*commit* or *abort*) of the memory to a $\blacktriangleright_i$ request immediately following σ.

A TS $\sigma \in \mathcal{T}$ is said to be *compatible* with the memory *TM* if:

1) For every prefix $\eta R_i(x,u)$ of σ, $\mathit{read}(\eta,i,x)=u$.
2) For every prefix $\eta \blacktriangleright_i$ of σ, $\mathit{close}(\eta,i)=\mathit{commit}$.
3) For every prefix $\eta \not\blacktriangleright_i$ of σ, $\mathit{close}(\eta,i)=\mathit{abort}$.

An implementation *TM*: $(\mathit{read},\mathit{close})$ is a *correct implementation of a transactional memory with respect to* $\mathcal{F}$ if every TS compatible with *TM* is also compatible with $\mathit{Spec}_{\mathcal{F}}$.

V. Verifying Implementation Correctness

In this section we present proof rules for verifying that an implementation satisfies the specification *Spec*. The approach is an adapted version of the rule presented in [4].

To apply the underlying theory, we assume that both the implementation and the specifications are represented as a *fair discrete system* (FDS) of the form $\mathcal{D}:\langle V,\mathcal{O},\Theta,\rho,\mathcal{J},\mathcal{C}\rangle$. We refer the reader to the appendix for additional details about this presentation of reactive systems.

In the current application, we prefer to adopt an *event-based* view of reactive systems, by which the observed behavior of a system is a (potentially infinite) set of events. Technically, this implies that the set of observable variables consists of a single variable $\mathcal{O}$, to which we refer as the *output variable*. It is also required that the domain of $\mathcal{O}$ always includes the value $\perp$, implying no observable event. In our case, the domain of the output variable is $E_\perp = E\cup\{\perp\}$.

Let $\eta: e_0,e_1,\ldots$ be an infinite sequence of $E_\perp$-values. The $E_\perp$-sequence $\widetilde{\eta}$ is called a *stuttering variant* of the sequence η if it can be obtained by removing or inserting finite strings of the form $\perp,\ldots,\perp$ at (potentially infinitely many) different positions within η.

Let $\sigma: s_0,s_1,\ldots$ be a computation of FDS $\mathcal{D}$. The *observation* corresponding to σ is the $E_\perp$ sequence $s_0[\mathcal{O}],s_1[\mathcal{O}],\ldots$ obtained by listing the values of the output variable $\mathcal{O}$ in each of the states. We denote by $\mathit{Obs}(\mathcal{D})$ the set of all observations of system $\mathcal{D}$.

Let $\mathcal{D}_C$ and $\mathcal{D}_A$ be two systems, to which we refer as the *concrete* and *abstract* systems, respectively. We say that system $\mathcal{D}_A$ *abstracts* system $\mathcal{D}_C$ (equivalently $\mathcal{D}_C$ *refines* $\mathcal{D}_A$), denoted $\mathcal{D}_C\sqsubseteq\mathcal{D}_A$ if, for every observation $\eta\in\mathit{Obs}(\mathcal{D}_C)$, there exists $\widetilde{\eta}\in\mathit{Obs}(\mathcal{D}_A)$, such that $\widetilde{\eta}$ is a stuttering variant of η. In other words, modulo stuttering, $\mathit{Obs}(\mathcal{D}_C)$ is a subset of $\mathit{Obs}(\mathcal{D}_A)$.

A. A Verification Rule Based on Abstraction Mapping

Based on the *abstraction mapping* of [5], we present in Fig. 2 a proof rule that reduces the abstraction problem into a verification problem. There, we assume two comparable FDS's, a *concrete* $\mathcal{D}_C:\langle V_C,\mathcal{O}_C,\Theta_C,\rho_C,\mathcal{J}_C,\mathcal{C}_C\rangle$ and an *abstract* $\mathcal{D}_A:\langle V_A,\mathcal{O}_A,\Theta_A,\rho_A,\mathcal{J}_A,\mathcal{C}_A\rangle$, and we wish to establish that $\mathcal{D}_C\sqsubseteq\mathcal{D}_A$. Without loss of generality, we assume that $V_C\cap V_A=\emptyset$, and that there exists a 1-1 correspondence between the concrete observables $\mathcal{O}_C$ and the abstract observables $\mathcal{O}_A$.

The method assumes the identification of an *abstraction mapping* $\alpha:(V_A=\mathcal{E}^\alpha(V_C))$ which assigns to each abstract variable $X\in V_A$ an expression $\mathcal{E}^\alpha_X$ over the concrete variables V_C. For an abstract assertion φ, we denote by $\varphi[\alpha]$ the assertion obtained by replacing each abstract variable $X\in V_A$ by its concrete expression $\mathcal{E}^\alpha_X$. We say that the abstract state S is an α-image of the concrete state s, written $S=\alpha(s)$, if the values of $\mathcal{E}^\alpha$ in s equal the values of the variables V_A in S.

Premise A1 of the rule states that if s is a concrete initial state, then $S=\alpha(s)$ is an initial abstract state.

Premise A2 states that if concrete state s_2 is a ρ_C-successor of concrete state s_1, then the abstract state $S_2=\alpha(s_2)$ is a ρ_A-successor of $S_1=\alpha(s_1)$. The box ( ) is the (linear time) temporal operator for "from here onwards." Together, A1 and A2 guarantee that, for every run $s_0,s_1,\ldots$ of $\mathcal{D}_C$ there exists a run $S_0,S_1,\ldots$ of $\mathcal{D}_A$, such that $S_j=\alpha(s_j)$ for every $j\geq 0$. Premise A3 states that the observables of the concrete state s and its α-image $S=\alpha(s)$ are equal. Premises A4 and A5 ensure that the abstract fairness requirements (justice and compassion, respectively) hold in any abstract state sequence which is a (point-wise) α-image of a concrete computation. Here,   is the (linear time) temporal operator for "eventually," thus,   means "infinitely often." It follows that every α-image of a concrete computation σ obtained by applications of premises A1 and A2 is an abstract computation whose observables match the observables of σ. This leads to the following claim:

Claim 2: If the premises of rule ABS-MAP are valid for some choice of α, then $\mathcal{D}_A$ is an abstraction of $\mathcal{D}_C$.

B. A Rule Based on an Abstraction Relation

It is not always possible to relate abstract to concrete states by a functional correspondence which maps each concrete state to a unique abstract state. In many cases, we cannot find an abstraction mapping, but can identify an *abstraction relation* $R(V_C,V_A)$ (which induces a relation $R(s,S)$).

In Fig. 3, we present proof rule ABS-REL which only assume an abstraction relation between the concrete and abstract states.

Premise R2 of the rule allows a single concrete transition to be emulated by a sequence of abstract transitions. This is done via the transitive closure ρ_A^+ which is defined as follows:

Let $S_0,S_1,\ldots,S_k$, $k>0$, be a sequence of abstract states, such that $\langle S_i,S_{i+1}\rangle\models\rho_A$ for every $i\in[0..k{-}1]$, and for some $\ell\in[1..k]$, for every $i\in[1..k]$, if $i\neq\ell$ then $S_i[\mathcal{O}]=\perp$. Then $\langle S_0,\widetilde{S}_k\rangle\models\rho_A^+$, where $\widetilde{S}_k=S_k[\mathcal{O}:=S_\ell[\mathcal{O}]]$ is obtained from S_k by assigning the variable $\mathcal{O}$ (the single output variable) the value that it has in state S_ℓ. This definition allows to perform first some "setting up" transitions that have no externally observable events, followed by a transition that produces a non-trivial observable value, followed by a finite number of "clean-up" transitions. The observable effect of the composite transition is taken to be the observable output of the only observable transition in the sequence.

Premise R1 of the rule states that for every initial concrete state s, it is possible to find an initial abstract state $S\models\Theta_A$, such that $\langle s,S\rangle\models R$.

Premise R2 states that for every pair of concrete states, s_1 and s_2, such that s_2 is a ρ_C-successor of s_1, and an abstract state S_1 which is a R-related to s_1, it is possible to find an abstract state S_2 such that S_2 is R-related to s_2 and is also a ρ_A^+-successor of S_1. Together, R1 and R2 guarantee that, for every run $s_0,s_1,\ldots$ of $\mathcal{D}_C$ there exists a run $S_0,\ldots,S_{i_1},\ldots,S_{i_2},\ldots,$ of $\mathcal{D}_A$, such that for every $j\geq 0$, S_{i_j} is R-related to s_j and all abstract states S_k, for $i_j<k<i_{j+1}$, have no observable variables. Premise R3 states that if abstract state S is R-related to the concrete state s, then the two states agree on the values of their observables. Premises R4 and R5 ensure that the abstract fairness requirements (justice and compassion, respectively) hold in any abstract state sequence which is a (point-wise) R-related to a concrete computation. It follows that every sequence of abstract states which is R-related to a concrete computation σ and is obtained by applications of premises R1 and R2 is an abstract computation whose observables match the observables of σ. This leads to the following claim:

Claim 3: If the premises of rule ABS-REL are valid for some choice of R, then $\mathcal{D}_A$ is an abstraction of $\mathcal{D}_C$.

VI. Transactional Memory Implementations

We now demonstrate how our proof rules can be used to verify three popular transactional memory implementations. Larus and Rajwar [2] classify transactional memory implementations in

$$\begin{array}{llll}
\textbf{A1}. & \Theta_C & \rightarrow & \Theta_A[\alpha] \\
\textbf{A2}. & \mathcal{D}_C & \models & (\rho_C \;\rightarrow\; \rho_A[\alpha][\alpha']) \\
\textbf{A3}. & \mathcal{D}_C & \models & (\mathcal{O}_C = \mathcal{O}_A[\alpha]) \\
\textbf{A4}. & \mathcal{D}_C & \models & J[\alpha], \qquad\qquad\qquad \text{for every } J \in \mathcal{J}_A \\
\textbf{A5}. & \mathcal{D}_C & \models & p[\alpha] \;\rightarrow\; q[\alpha], \quad \text{for every } (p,q) \in \mathcal{C}_A \\
\hline
 & \multicolumn{3}{c}{\mathcal{D}_C \sqsubseteq \mathcal{D}_A}
\end{array}$$

Fig. 2. Rule ABS-MAP.

$$\begin{array}{llll}
\textbf{R1}. & \Theta_C & \rightarrow & \exists V_A : R \wedge \Theta_A \\
\textbf{R2}. & \mathcal{D}_C & \models & (R \wedge \rho_C \;\rightarrow\; \exists V'_A : R' \wedge \rho_A^{+}) \\
\textbf{R3}. & \mathcal{D}_C & \models & (R \;\rightarrow\; \mathcal{O}_C = \mathcal{O}_A) \\
\textbf{R4}. & \mathcal{D}_C & \models & (\forall V_A : R \rightarrow J), \qquad\qquad\qquad \text{for every } J \in \mathcal{J}_A \\
\textbf{R5}. & \mathcal{D}_C & \models & (\exists V_A : R \wedge p) \;\rightarrow\; (\forall V_A : R \rightarrow q), \quad \text{for every } (p,q) \in \mathcal{C}_A \\
\hline
 & \multicolumn{3}{c}{\mathcal{D}_C \sqsubseteq \mathcal{D}_A}
\end{array}$$

Fig. 3. Rule ABS-REL.

terms of several properties. We focus on two of these properties, *conflict detection* and *version control*, both of which can be either "eager" or "lazy," depending when conflicts are detected and when the memory is updated. Since one cannot have eager version management with lazy conflict detection, there are three possibilities left. We give a detail description of the proof of the lazy conflict detection and lazy version control, and sketch the remaining two.

A. Lazy Conflict Detection, Lazy Version Control

Denote this class by **ll**. A representative of this class is TCC [6], and we give a simple implementation from this class that we refer to as TM_1.

The implementation uses the following data structures:

- *imp_mem*: $\mathbb{N} \;\rightarrow\; \mathbb{N}$ — A persistent memory. Initially, *imp_mem*$[j] = 0$ for all $j \in \mathbb{N}$;
- *trans*: **array**$[1..n]$ **of list of** E — An array of lists. For each $i \in [1..n]$, *trans*$[i]$ is a sequence over E_i that lists the events of the currently pending transaction of client i, if such exists. Initially, every *trans*$[i]$ is empty;
- *imp_out*: scalar in $E_\perp = E \cup \{\perp\}$ — an output variable recording responses to clients, initially $\perp$.

The implementation reacts to possible requests by the clients. It accepts a request of $\blacktriangleleft_i$ ("open transaction"), and rejects any other request if *trans*$[i]$ is empty. An accepted $R_i(x)$ request is responded by u, where u is such that $W_i(x,u)$ is the last $W_i(x)$ event in *trans*$[i]$, or, if no such event exists, by *imp_mem*$[x]$; Upon an accepted $\blacktriangleright_i$ request, TM_1 checks whether the transaction *trans*$[i]$ is consistent with *imp_mem*. If it is, TM_1 returns to Client i a "commit", updates *imp_mem* according to *trans*$[i]$, and resets *trans*$[i]$ to be empty. If *trans*$[i]$ is not consistent with *imp_mem*, TM_1 returns an "abort," and resets *trans*$[i]$ to empty.

Finally, the events corresponding to accepted requests are written to *imp_out*, which is set to $\perp$ with steps that don't produce a response. Each of these events (with the exception of $\blacktriangleright$ and $\not\blacktriangleright$), is appended to the appropriate *trans*$[i]$.

The specification, described in Section IV, specifies not only the behavior of the Transactional Memory but also the combined behavior of the memory when coupled with a typical clients module. A generic clients module, *Clients*(n), may, at any step, issue the next request for client i, $i \in [1..n]$, provided the sequence of E_i-events issued so far (including the current one) forms a prefix of a well-formed TS. The justice requirement of *Clients*(n) is that eventually, every open transaction must be closed by issuing a $\blacktriangleright_i$-request.

Combining modules TM_1 and *Clients*(n) we obtain the complete implementation, defined by:

$$Imp_1 : \quad TM_1 \;|||\; Clients(n)$$

where $|||$ denote the *synchronous* composition operator defined in the appendix. We interpret this composition in a way that combines several of the actions of each of the modules into a single transition.

The possible actions of Imp_1 are the following:

t_1. Set *imp_out* $=$ *trans*$[i]$ $= \blacktriangleleft_i$ if *trans*$[i] = \Lambda$;
t_2. Set *imp_out* to $R_i(x,u)$ and append it to *trans*$[i]$ if *trans*$[i]$ is non-empty, and the last $W_i(x)$ event in it is $W_i(x,u)$, or if *trans*$[i]$ contains no $W_i(x)$ event and $u =$ *imp_mem*$[x]$;
t_3. Set *imp_out* to $W_i(y,v)$ and append $W_i(y,v)$ to the end of *trans*$[i]$ if *trans*$[i]$ is non-empty;
t_4. Set *imp_out* to $\blacktriangleright_i$, update *imp_mem* according to *trans*$[i]$, and reset *trans*$[i]$ to empty if *trans*$[i]$ is non-empty and consistent with *imp_mem*;
t_5. Set *imp_out* to $\not\blacktriangleright_i$ and set *trans*$[i]$ to empty if *trans*$[i]$ is non-empty and is inconsistent with *imp_mem*;
t_6. Set *imp_out* to $\perp$ and leave all other variables unchanged.

Since *Clients*(n)'s justice requires every transaction to eventually issue a $\blacktriangleright$ request, and since t_4 and t_5 guarantee that each $\blacktriangleright$ request empties the corresponding *trans*$[i]$, it follows that module Imp_1 has a justice requirement: for each $i = 1, \ldots, n$, *trans*$[i]$ is empty infinitely many times.

We now sketch a proof, using Rule ABS-REL, that $Imp_1 \sqsubseteq Spec$.

The application of rule ABS-REL requires the identification of a relation R which holds between concrete and abstract states. We use the relation R defined by:

$$spec\_out = imp\_out \wedge spec\_mem = imp\_mem \wedge \bigwedge_{i=1}^{n} (q|_i = trans[i])$$

The relation R stipulates equality between *spec_out* and *imp_out* – the output of the implementation, and between *spec_mem* and *imp_mem*, and that, for each $i \in [1..n]$, the projection of q on the set of events pertinent to Client i equals *trans*$[i]$.

To simplify the proof, we assume (see the end of Section II) that all transactions have the form

$$\blacktriangleleft_i \; R_i(x_1,u_1)\cdots R_i(x_r,u_r) W_i(y_1,v_1)\cdots W_i(y_w,v_w) \{\blacktriangleright_i, \not\blacktriangleright_i\}$$

It is not difficult to see that premise R1 of rule ABS-REL holds, since the two initial conditions are given by

$$\begin{array}{ll}\Theta_C: & \textit{imp\_out} = \bot \ \wedge\ \textit{imp\_mem} = \lambda i.0 \ \wedge\ \bigwedge_{i=1}^{n}(\textit{trans}[i] = \Lambda) \\ \Theta_A: & \textit{spec\_out} = \bot \ \wedge\ \textit{spec\_mem} = \lambda i.0 \ \wedge\ q = \Lambda\end{array}$$

and the relation R guarantees equality between the relevant variables.

The R-conjunct *spec_out* $=$ *imp_out* guarantees the validity of premise R3.

We will now examine the validity of premise R2. This can be done by considering each of the concrete transitions $t_1, \ldots, t_6$.

- t_1. Transition t_1 appends the event $\blacktriangleleft_i$ to an empty *trans*$[i]$ and outputs it to *imp_out*. This can be emulated by an instance of abstract transition a_1 which output $\blacktriangleleft_i$ to *spec_out* and places this event at the end of q. It can be checked that this joint action preserves the relation R, in particular, the relevant conjunct $\bigwedge_{j=1}^{n}(q|_j = \textit{trans}[j])$.
- t_2. Transition t_2 appends to *trans*$[i]$ (and outputs) the event $R_i(x, u)$ where, due to the simplifying assumption, $u =$ *imp_mem*$[x]$. This can be matched by another instance of abstract transition a_3.
- t_3. Transition t_3 appends to *trans*$[i]$ (and outputs) the event $W_i(y, v)$, which is matched by an instance of abstract transition a_2.
- t_4. Transition t_4 closes and commits the current transaction contained in *trans*$[i]$ while outputting the event $\blacktriangleright_i$. This is possible if the transaction pending in *trans*$[i]$ is consistent with *imp_mem*. The transition also updates *imp_mem* according to *trans*$[i]$, and then clears *trans*$[i]$.
  The emulation of this transition begins by the instance of a_5 which appends $mark_i$ to q, followed by a sequence of applications of abstract transition a_8 which attempts to move all the elements of *trans*$[i]$ to the front of the queue q, where $\mathcal{F}$ is the trivial predicate F (thus, allowing any interchange). If successful, we apply abstract transition a_7 which confirms that *trans*$[i]$ is consistent with *spec_mem* (must be true due to the R-conjunct *spec_mem* $=$ *imp_mem*), updates *spec_mem* according to *trans*$[i]$ (thus making it again equal to *imp_mem*), and removes all elements of *trans*$[i]$ from q, thus reestablishing the R-conjunct $\bigwedge_{j=1}^{n}(q|_j = \textit{trans}[j])$.
- t_5. Transition t_5 closes and aborts the transaction pending in *trans*$[i]$ while outputting the event $\not\blacktriangleright_i$. This is possible only if the transaction pending in *trans*$[i]$ is inconsistent with *imp_mem*. The transition also clears *trans*$[i]$.
  The transition t_5 is matched with the abstract transition a_6 which outputs the event $\not\blacktriangleright_i$ and removes from q all elements of the aborted transition *trans*$[i]$. Note that *Spec* does not require an aborted transaction to be "uncommitable," thus, we don't have to (though we can) ensure that *Spec* cannot commit *trans*$[i]$.
- t_6. The idling concrete transition t_6 may be emulated by the idling abstract transition a_9.

It remains to verify premise R4. This premise requires showing that any concrete computation visits infinitely many times states satisfying $\forall V_A : R \rightarrow J_A$, where $J_i : q|i = \Lambda$, characterizes the set of abstract states in which the queue contains no E_i event. Since R requires that $q|_i =$ *trans*$[i]$, we obtain that Premise R4 is valid.

Premise R5 is vacuously valid since *Spec* has no compassion requirements.

Note that ABS-MAP does not suffice to construct step t_4, where the power of ABS-REL is demonstrated. We obtained a similar proof for a bounded instantiation using TLC, however, there *Spec* is defined as performing "meta-steps," without which TLC, that uses an ABS-MAP-like rule, cannot construct the relations ABS-REL does.

B. *Eager Conflict Detection, Lazy Version Control*

Denote this class by **el**. A representative of **el** is LTM of [7]. Its definition of "conflict" is slightly stronger than "eager invalidation" by having writes to the same object as a conflict, thus, its forbidden interchange set consists of $\mathcal{F}_{ei}$ and all pairs of the form $(W_i(x), W_j(x))$. In case of a conflict, the transaction that requests the second "offensive" memory access is aborted.

The main difference between **el** and the prior implementation $\textit{TM}_1$ is the conflict detection: upon receiving a $R_i(x)$ such that $W_j(x)$ is in some open transaction, or a $W_i(x, v)$ such that $W_j(x)$ or $R_j(x)$ is in some open transaction, the transaction of client i is aborted. The system performs two steps – the first returns the result of the operation, and the second aborts the transaction. Thus, an abort is not only a possible response to a non-close transaction request, but every transaction that requests to be closed is committed. For our higher level description of this implementation, we add a new variable *toabort* $\in [0..n]$, that holds the id of the client whose transaction is to be aborted (0 indicates no such client exists).

The combination of an **el** memory and *Clients*(n) is the module **Iel** whose possible actions are:

- t_1. If *toabort* $= i > 0$, then set *imp_out* to $\not\blacktriangleright_i$, empty *trans*$[i]$, and set *toabort* to 0;

Else, do one of the following:

- t_2. Set *imp_out* $=$ *trans*$[i] = \blacktriangleleft_i$ if *trans*$[i]$ is empty;
- t_3. Set *imp_out* to $R(x, u)$, and append it to *trans*$[i]$, if *trans*$[i] \neq \Lambda$, $u =$ *imp_mem*$[x]$ or $W_i(x) \in$ *trans*$[i]$ and the most recent such event is $W_i(x, u)$, and for every $j \neq i$, $W(x) \notin$ *trans*$[j]$, ;
- t_4. Set *imp_out* to $R(x, \textit{imp\_mem}[u])$, append it to *trans*$[i]$, and set *toabort* to i if *trans*$[i] \neq \Lambda$, and for some $j \neq i$, or $W_j(x) \in$ *trans*$[j]$;
- t_5. Set *imp_out* to $W_i(x, v)$ and append it to *trans*$[i]$, if *trans*$[i] \neq \Lambda$ and for every $j \neq i$, $W(x), R(x) \notin$ *trans*$[j]$;
- t_6. Set *imp_out* to $W_i(x, v)$, append it to *trans*$[i]$, and set *toabort* to i, if *trans*$[i] \neq \Lambda$ and for some $j \neq i$, $R_j(x)$ or $W_j(x) \in$ *trans*$[j]$;
- t_7. Set *imp_out* to $\blacktriangleright_i$, update *imp_mem* according to *trans*$[i]$ and reset *trans*$[i]$ to Λ, if *trans*$[i]$ is not empty;
- t_8. Set *imp_out* to $\bot$ and leave all other variables unchanged;

Module **Iel** has a justice requirement for each $i = 1, \ldots, n$, requiring that *trans*$[i] = \Lambda$ infinitely many times.

To prove that **Iel** satisfies the specifications of Section IV, we use the same R used to verify $\textit{TM}_1$, with respect to the admissible interchange associated with **el**.

STM of [8] is also an **el** implementation. There, clients must first obtain write locks on all memory locations they are likely to access in a transaction (the locks are requested in increasing order, to avoid deadlocks), which are released when the transaction completes. The locking mechanism can be accomplished by adding to each memory location an "owner" in the range $[0..n]$ indicating which client currently has a write-lock on it, and refining **Iel** to accommodate the needs of STM.

C. *Eager Conflict, Eager Version Control*

Denote this class by **ee**. A representative of **ee** is LogTM of [9]. Its definition of "conflict" and their resolution are exactly like those of **el**. Being eager-version, however, **ee** protocols update the memory upon a write. If later it is necessary to abort the transaction, then the memory is rolled back to its previous value. Since the protocol does not allow for more than one overlapping write, there is no need to record any information but the previous value of W's in pending transactions. To thus add a set $\mathit{committed} \subseteq n \times \mathbb{N} \times \mathbb{N}$ where n is a client id. $\mathit{committed}$ stores, for every memory address x that was written by a currently pending transaction, the previous value written to it (by a committed transaction). Initially, $\mathit{committed} = \emptyset$.

The combined implementation of **ee** memory and $\mathit{Clients}(n)$ is the module **Iee** whose possible actions are similar to that of **Iel**, but for t_1, t_5 and t_7, that are now:

t_1. If $\mathit{toabort} = i > 0$, then
1) set $\mathit{imp\_out}$ to $\not\blacktriangleright_i$;
2) for every $(i, x, v) \in \mathit{committed}$, set $\mathit{imp\_mem}[x]$ to v and remove (i, x, v) from $\mathit{committed}$; set $\mathit{trans}[i] = \Lambda$ and $\mathit{toabort} = 0$;

t_5. Set $\mathit{imp\_out}$ to $W_i(x, v)$, append it to $\mathit{trans}[i]$, add $(i, x, \mathit{imp\_mem}[x])$ to $\mathit{committed}$, and set $\mathit{imp\_mem}[x]$ to v, if $\mathit{trans}[i] \neq \Lambda$, and for every $j \neq i$, $W(x), R(x) \notin \mathit{trans}[j]$;

t_7. Set $\mathit{imp\_out}$ to $\blacktriangleright_i$, reset $\mathit{trans}[i]$ to Λ and remove from $\mathit{committed}$ every (i, x, v), if $\mathit{trans}[i]$ is not empty;

Module **Iee** has the same justice requirement as its predecessors.

To prove that **Iee** satisfies the specifications of Section IV, we cannot use the same R used to verify TM_1; rather, we look at the "rolled back" version of memory values, which can be determined by $\mathit{committed}$. Formally, for each memory address $x \in \mathbb{N}$, we define

$$\mathit{rolled\_back}[x] \;=\; \begin{cases} v & \text{for some } j, \\ & (j, x, v) \in \mathit{committed} \\ \mathit{imp\_mem}[x] & \text{otherwise} \end{cases}$$

For the memory $\mathit{imp\_mem}$, $\mathit{rolled\_back}(\mathit{imp\_mem})$ is $\mathit{imp\_mem}$ where every entry is replaced by its rollback entry. Then the relation R_{STM} is defined by:

$$R_{\text{STM}}: \qquad \mathit{spec\_out} = \mathit{imp\_out} \;\wedge\; \bigwedge_{i=1}^{n} (q|_i = \mathit{trans}[i]) \;\wedge \\ \mathit{spec\_mem} = \mathit{rolled\_back}(\mathit{imp\_mem})$$

VII. Verification with TLC

We verified the correctness of all implementations above by the explicit-state model checker TLC, the input of which are TLA$^+$ programs. See [10] for a thorough discussion of TLC and TLA$^+$. Based on the similarity between TLC and the FDS model, we verified that all the implementations above indeed implement our trivial specification of Section IV.

To verify that an implementation correctly implements its specification, one has to provide TLA$^+$ modules for both specification and implementation, and a mapping associating each of the specification's variables with an expression over the implementation's variables. With these, TLC verifies that the mapping is a refinement mapping satisfying the premises of Rule ABS-MAP. (In fact, the rule TLC uses is somewhat different, but suffices for our needs.) Since TLC can handle only finite-state systems, all parameters – memory size, number of clients, bound on pending transactions, etc. – have to be bounded.

A. Specification Module

The specification module is constructed from two submodules, *Spec* and *Driver*. Submodule *Spec* is the core of the specification and is uniform for all *TM* specifications. It is essentially the specification module of Section IV. *Driver* defines features that are unique to each transactional memory by means of a resolving predicate $\mathcal{F}$. *Driver* can only restrict the next state relation and cannot introduce new transitions that are not defined in Section IV.

B. Implementation Module

All implementations include a module *Imp* that consists of a synchronous composition of the memory and the clients, such that every request by a client is immediately responded by the memory.

Since TLC requires that every *Spec* variable has a matching expression over *Imp* variables, we added a new variable to *Imp*, $\mathit{history\_q}$, which is a queue over $E_\perp$ that contains all events of pending transactions. New events are appended to $\mathit{history\_q}$, and the events of a transaction that is closed (committed or aborted) are removed from it.

C. Refinement mapping

The implementation module includes a mapping between *Spec*'s variables, *spec_mem*, q, *spec_out*, and *doomed*, to expressions over *Imp*'s variables. In all but our last example the refinement mapping is trivial: $\mathit{spec\_mem} = \mathit{imp\_mem}$, $q = \mathit{history\_q}$, $\mathit{spec\_out} = \mathit{imp\_out}$, and $\mathit{doomed}[i] = \text{F}$ for all i. In the last example, $\mathit{spec\_mem} = \mathit{rolled\_back}(\mathit{imp\_mem})$ replaces $\mathit{spec\_mem} = \mathit{imp\_mem}$. TLC (automatically) verifies that the proposed mapping is a refinement mapping. Success means that, for the bounded instantiation taken, *Imp* implements its specification *Spec*, i.e., that every *Imp* implements some *Spec* run, and that every fair *Imp* computation maps into a fair *Spec* computation. In the first case, failure is indicated by a finite execution path leading from an initial state into a state in which the mapping is falsified. In the second case, failure is indicated by a finite execution path leading from an initial state to a loop in which the implementation meets all fairness requirements, and the associated specification does not.

VIII. Conclusion and Future Work

In this paper we developed a formal specification of transactional memory correctness and a methodology for verifying transactional memory implementations based on model checking. We demonstrated our approach on three transactional memory implementations drawn from the literature. While our models capture the important algorithmic aspects of those implementations, they are still quite a bit more abstract than "real" implementations in the form of C++ or Java libraries, say. The most obvious next step is to formally analyze more detailed models of implementations.

Practical transactional memory implementations must deal with memory accesses that occur outside of transactions. Such non-transactional accesses give rise to anomalies like the *privatization problem* [11], in which a thread can observe inconsistencies in what should be its own private copy of some shared data; and the *granular lost update problem* [12], in which the transactional implementation manages memory at a coarser granularity than changes made by nontransactional updates, leading to nontransactional updates being lost. It would be interesting to extend our formal specification and verification framework to account for non-transactional accesses and give precise and abstract characterizations of the privatization and GLU problems.

There are also a number of open questions concerning the programmer's view of transactions, and we want to extend our framework to reason about them, too. For example,

- What happens when transactions contain other transactions? Two kinds of transaction nesting have been proposed: in *closed nesting* the nested transactions are "flattened" into one top-level transaction whose effects are invisible until commit time, while in *open nesting* the effects of nested transactions may be visible before commit. In open nesting the requirement for serializability is relaxed, and it would be interesting to extend our specification to account for this.

- What are the properties of various linguistic constructs for programming with transactions? This is an active area of research in the programming languages community (see [13] for one example).

Finally, we would like to harness the power of new verification technology like satisfiability modulo theories (SMT) that has already shown so much potential for software verification. Interesting questions are whether SMT and other software verification technology gives us additional leverage for efficient reasoning about transactional memory, and whether there are theories and decision procedures specific to transactional memory that we could add to the SMT arsenal.

References

[1] M. Herlihy and J. E. B. Moss, "Transactional memory: architectural support for lock-free data structures," in *ISCA '93: Proceedings of the 20th annual international symposium on Computer architecture*. New York, NY, USA: ACM Press, 1993, pp. 289–300.

[2] J. R. Larus and R. Rajwar, *Transactional Memory*. Morgan & Claypool Publishers, 2007.

[3] M. Scott, "Sequential specification of transactional memory semantics," in *Proc. TRANSACT the First ACM SIGPLAN Workshop on Languages, Compiler, and Hardware Suppport for Transactional Computing*, Ottawa, 2006.

[4] Y. Kesten, A. Pnueli, E. Shahar, and L. D. Zuck, "Network invariants in action," in *13th International Conference on Concurrency Theory (CONCUR02)*, ser. Lect. Notes in Comp. Sci., vol. 2421. Springer-Verlag, 2002, pp. 101–115.

[5] M. Abadi and L. Lamport, "The existence of refinement mappings," *Theoretical Computer Science*, vol. 82, no. 2, pp. 253–284, May 1991.

[6] L. Hammond, W.Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Herzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun, "Transactional memory coherence and consistency," in *Proc. 31^{s}t annu. Int. Symp. on COmputer Architecture*, Jun. 2004.

[7] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie, "Unbounded transactional memory," in *Proceedings of the Eleventh International Symposium on High-Performance Computer Architecture*, Feb 2005, pp. 316–327.

[8] N. Shavit and D. Touitou, "Software transactional memory," in *Proc. 14th ACM Symp. Princ. of Dist. Comp.* Ottawa Ontario CA: ACM Press, 1995, pp. 204–213.

[9] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood, "Logtm: Log-based transactional memory," in *Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, Feb 2006, pp. 254–265.

[10] L. Lamport, *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[11] M. F. Spear, V. J. Marathe, L. Dalessandro, and M. L. Scott, "Privatization techniques for software transactional memory," Department of Computer Science, University of Rochester, Tech. Rep. 915, Feb. 2007.

[12] T. Shpeisman, V. Menon, A.-R. Adl-Tabatabai, S. Balensiefer, D. Grossman, R. Hudson, K. F. Moore, and B. Saha, "Enforcing isolation and ordering in STM," in *ACM Conference on Programming Language Design and Implementation*, Jun. 2007.

[13] T. Harris, S. Marlow, S. P. Jones, and M. Herlihy, "Composable memory transactions," in *PPoPP*, Jun. 2005.

[14] Y. Kesten and A. Pnueli, "Control and data abstractions: The cornerstones of practical formal verification," *Software Tools for Technology Transfer*, vol. 4, no. 2, pp. 328–342, 2000.

[15] Z. Manna and A. Pnueli, *Temporal Verification of Reactive Systems: Safety*. New York: Springer-Verlag, 1995.

Appendix

Fair Discrete Systems and Their Computations As a computational model for reactive systems we take the model of *fair discrete systems* (FDS) [14], which is a slight variation on the model of *fair transition system* [15]. Under this model, a system $\mathcal{D} : \langle V, \mathcal{O}, \Theta, \rho, \mathcal{J}, \mathcal{C}\rangle$ consists of the following components:

- V — A set of *system variables*. A *state* of $\mathcal{D}$ provides a type-consistent interpretation of the variables V. For a state s and a system variable $v \in V$, we denote by $s[v]$ the value assigned to v by the state s. Let Σ denote the set of all states over V.
- $\mathcal{O} \subseteq V$ — A subset of *observable variables*. These are the variables which can be externally observed.
- Θ — The *initial condition*: An assertion (state formula) characterizing the initial states.
- $\rho(V, V')$ — The *transition relation*: An assertion, relating the values V of the variables in state $s \in \Sigma$ to the values V' in an $\mathcal{D}$-successor state $s' \in \Sigma$. We assume that every state has a ρ-successor.
- $\mathcal{J}$ — A set of *justice* (*weak fairness*) requirements (assertions); A computation must include infinitely many states satisfying each of the justice requirements.
- $\mathcal{C}$ — A set of *compassion (strong fairness)* requirements: Each compassion requirement is a pair $\langle p, q\rangle$ of state assertions; A computation should include either only finitely many p-states, or infinitely many q-states.

For an assertion ψ, we say that $s \in \Sigma$ is a ψ-state if $s \models \psi$.

A *run* of an FDS $\mathcal{D}$ is a possibly infinite sequence of states $\sigma : s_0, s_1, \ldots$ satisfying the requirements:

- *Initiality* — s_0 is initial, i.e., $s_0 \models \Theta$.
- *Consecution* — For each $\ell = 0, 1, \ldots$, the state $s_{\ell+1}$ is an $\mathcal{D}$-successor of s_ℓ. That is, $\langle s_\ell, s_{\ell+1}\rangle \models \rho(V, V')$ where, for each $v \in V$, we interpret v as $s_\ell[v]$ and v' as $s_{\ell+1}[v]$.

A *computation* of $\mathcal{D}$ is an infinite run that satisfies

- *Justice* — for every $J \in \mathcal{J}$, σ contains infinitely many occurrences of J-states.
- *Compassion* – for every $\langle p, q\rangle \in \mathcal{C}$, either σ contains only finitely many occurrences of p-states, or σ contains infinitely many occurrences of q-states.

A *synchronous parallel composition* of systems $\mathcal{D}_1$ and $\mathcal{D}_2$, denoted by $\mathcal{D}_1 ||| \mathcal{D}_2$, is specified by the FDS

$$\mathcal{D} : \langle V_1 \cup V_2, \mathcal{O}_1 \cup \mathcal{O}_2, \Theta_1 \wedge \Theta_2, \rho_1 \wedge \rho_2, \mathcal{J}_1 \cup \mathcal{J}_2, \mathcal{C}_1 \cup \mathcal{C}_2\rangle$$

To guarantee that the composition doesn't cause any computation of the composed system to be lost, we further require that for every $i = 1, 2$, each $\mathcal{D}_i$-computation is some computation of $\mathcal{D}$ when projected onto V_i.