Software Transactional Memory on Relaxed Memory Models*

Rachid Guerraoui, Thomas A. Henzinger, and Vasu Singh

EPFL, Switzerland

Abstract. Pseudo-code descriptions of STMs assume sequentially consistent program execution and atomicity of high-level STM operations like read, write, and commit. These assumptions are often violated in realistic settings, as STM implementations run on relaxed memory models, with the atomicity of operations as provided by the hardware. This paper presents the first approach to verify STMs under relaxed memory models with atomicity of 32 bit loads and stores, and read-modify-write operations. We present RML, a new high-level language for expressing concurrent algorithms with a hardware-level atomicity of instructions, and whose semantics is parametrized by various relaxed memory models. We then present our tool, FOIL, which takes as input the RML description of an STM algorithm and the description of a memory model, and automatically determines the locations of fences, which if inserted, ensure the correctness of the STM algorithm under the given memory model. We use FOIL to verify DSTM, TL2, and McRT STM under the memory models of sequential consistency, total store order, partial store order, and relaxed memory order.

1 Introduction

Software transactional memory (STM) [11,19] is now widely accepted as a concurrent programming model. STM allows the programmer to think in terms of coarse-grained code blocks that appear to be executed atomically, and at the same time, yields a high level of parallelism. The algorithm underlying an STM is non-trivial, precisely because it simplifies the task of the programmer by encapsulating the difficulty of synchronization and recovery. Various correctness criteria have been proposed for STM algorithms. One criterion, popular for its relevance to the STM designers, is opacity [8]. Opacity is motivated by the fact that in STMs, observing inconsistent state by even an aborted transaction can lead to unexpected side effects. Opacity builds upon strict serializability [17], a correctness property used for database transactions. Strict serializability requires that the committed transactions appear to be executed in a serial order, consistent with the order of non-overlapping transactions. Opacity further requires that even aborted transactions appear to be executed in a serial order.

Previous attempts at formally verifying the correctness of STMs [3,6,7] with respect to different correctness criteria assumed that high-level transactional commands like start, read, write, commit, and abort execute atomically and in a sequentially consistent manner. Verification of an STM at this level of abstraction leaves much room for

* This research was supported by the Swiss National Science Foundation.

A. Bouajjani and O. Maler (Eds.): CAV 2009, LNCS 5643, pp. 321–336, 2009.
© Springer-Verlag Berlin Heidelberg 2009

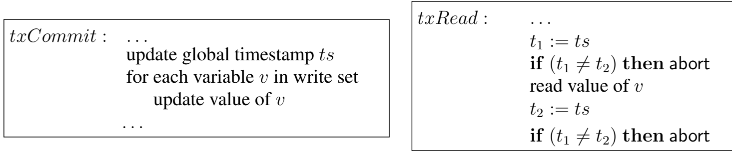

Fig. 1. Code fragments of commit and read procedures of a timestamp-based STM

errors in a realistic setting. This is because the actual hardware on which STMs run supports a finer-grained degree of atomicity: in practice, the set of atomic instructions rather corresponds to *load*, *store*, and *read-modify-write*. Furthermore, compilers and processors assume relaxed memory models [1] and are notorious for playing tricks to optimize performance, e.g., by reversing the order of instructions to different addresses. Typically, STM designers use *fences* to ensure a strict ordering of memory operations. As fences hurt performance, STM designers want to use fences only when necessary for correctness.

To illustrate some of the issues, consider the code fragments of the commit and the read procedures of a typical timestamp-based STM like TL2 [4] in Figure 1. Assume that at the start of a transaction, t_1 and t_2 are set to the global timestamp *ts*. The commit procedure updates the timestamp *ts* before it updates the variables in the write set. The read procedure first reads the timestamp, followed by the read of the variable, followed by a second read of the timestamp. The read is successful only if the two timestamps are equal. A crucial question is, given the memory model, which fences are required to keep the STM correct. On a memory model like sequential consistency [12] or total store order [20], the code fragment in Figure 1 is correct without fences. On the other hand, on memory models that relax store order, like partial store order [20], we need to add a *store fence* after the timestamp update in the commit procedure. For even more relaxed memory models that may swap independent loads, like relaxed memory order [20], as well as the Java memory model [16], we need more fences, namely, *load fences* in the read procedure. But the question is how many? Do we need to ensure that the read of v is between the two reads of *ts*, and thus put two fences? The answer is no. To ensure correctness, we just need one fence and guarantee that the second read of *ts* comes after the read of v.

Devising a verification technique to model check STM algorithms assuming relaxed memory models and hardware-level atomicity is challenging. A first challenge is to devise a precise and unified formalism in which the STM implementations can be expressed. A second challenge is to cope with the non-determinism explosion. Not surprisingly, when compared to verifying an STM at a high-level atomic alphabet, the level of non-determinism to be dealt with at hardware-level atomicity under a relaxed memory model is much higher. For example, the implementation of DSTM [10] with 2 threads and 2 variables generates 1,000 states with a high-level atomic alphabet [6] and 1,200,000 states with a low-level one, even on a sequentially consistent memory model. A relaxed memory model further increases the state space.

This paper takes up the challenge of bridging the gap between STM descriptions in the literature and their real implementations on actual hardware. We start by presenting

a formalism to express memory models as a function of hardware memory instructions, that is, loads and stores to 32 bit words. We describe various relaxed memory models, such as total store order (TSO), partial store order (PSO), and relaxed memory order (RMO) in our formalism. The reason for choosing these memory models is to capture different levels of relaxations allowed by different multiprocessors. Unlike earlier formalisms [3,6] used for verification, our formalism can be used to express and check the correctness of STMs with both update semantics: direct (eager) and deferred (lazy). Then, we present a new language, RML (*Relaxed Memory Language*), with a hardware-level of atomicity, whose semantics is parametrized by various relaxed memory models. At last, we describe a new tool, FOIL (a fencing weapon), to verify the opacity of three different STM algorithms, DSTM, TL2, and McRT STM, under different memory models. We choose these STMs as they represent three different and important trends in STM design. DSTM is obstruction-free (does not use locks), TL2 is a lock-based STM with deferred-update semantics, and McRT STM is a lock-based STM with direct-update semantics. While we choose opacity as the correctness criterion, using FOIL we can also verify other correctness properties such as strict serializability that can be specified in our formalism.

FOIL proves the opacity of the considered STM algorithms under sequential consistency and TSO. As the original STM algorithms have no fences, FOIL generates counterexamples to opacity for the STMs under further relaxed memory models (PSO and RMO), and automatically inserts fences within the RML description of the STM algorithms which are required (depending upon the memory model) to ensure opacity. We observe that FOIL inserts fences in a pragmatic manner, as all fences it inserts match those in the manually optimized official implementations of the considered STMs. Our verification leads to an interesting observation that many STMs are sensitive to the order of loads and stores, but neither to the order of a store followed by a load, nor to store buffering. Thus, while all STM algorithms we consider need fences for opacity under PSO and RMO, they are indeed opaque under TSO without any fences.

2 Framework

We first present a general formalism to express hardware memory instructions and memory models. Then, we formalize the correctness of STMs at the level of hardware instructions.

Memory instructions. Let $Addr$ be a set of memory addresses. Let I be the set of *memory instructions* that are executed atomically by the hardware. We define the set I as follows, where $a \in Addr$:

$$I ::= \langle \mathsf{load}\ a \rangle \mid \langle \mathsf{store}\ a \rangle \mid \langle \mathsf{cas}\ a \rangle$$

We use the $\langle \mathsf{cas}\ a \rangle$ instruction as a generic read-modify-write instruction.

Memory models. Memory models [1] specify the behavior of memory instructions to shared memory in a multiprocessor setting. For example, the memory model *sequential consistency* specifies that a multiprocessor executes the instructions of a thread in program order. On the other hand, the memory model *total store order* specifies that a multiprocessor may relax the order of a store followed by a load to a different address,

by delaying stores using a store buffer. In principle, a memory model offers a tradeoff between transparency to the programmer and flexibility to the hardware to optimize performance. Sequential consistency is the most stringent memory model, and thus the most intuitive to the programmer. But, most of the available multiprocessors do not support sequential consistency for reasons of performance. We present a formalism to express relaxed memory models.

A *memory model* is a function $M : I \times I \rightarrow \{N, E, Y\}$. For all instructions $i, j \in I$, when i is immediately followed by j, we have: (i) if $M(i, j) = N$, then M imposes a strict order between i and j, (ii) if $M(i, j) = E$, then M allows to eliminate the instruction j, (iii) if $M(i, j) = Y$, then M allows to reorder i and j. The case (ii) allows us to model store load forwarding using store buffers, and case (iii) allows us to model reordering of instructions. Thus, our formalism can capture many of the hardware memory models. But, our formalism cannot capture some common compiler optimizations like irrelevant read elimination, and thus disallows many software memory models (like the Java memory model [16]). We specify different memory models in our framework. These memory models are chosen to illustrate different levels of relaxations generally provided by the hardware.

1. Sequential consistency does not allow any pair of instructions to be reordered. *Sequential consistency* [12] is specified by the memory model M_{sc}. We have $M_{sc}(i, j) = N$ for all instructions $i, j \in I$.

2. Total store order (TSO) relaxes the order of a store followed by a load to a different address. But, the order of stores cannot be reordered. TSO allows a load which follows a store to the same address to be eliminated. TSO [20] is given by the memory model M_{tso} such that for all memory instructions $i, j \in I$, (i) if $i = \langle \mathsf{store}\ a \rangle$ and $j = \langle \mathsf{load}\ a' \rangle$ such that $a \neq a'$, then $M_{tso}(i, j) = Y$, (ii) if $i \in \{\langle \mathsf{store}\ a \rangle, \langle \mathsf{cas}\ a \rangle\}$ and $j = \langle \mathsf{load}\ a \rangle$, then $M_{tso}(i, j) = E$, (iii) else $M_{tso}(i, j) = N$.

3. Partial store order (PSO) is similar to TSO, but further relaxes the order of stores. PSO [20] is specified by M_{pso}, such that for all memory instructions $i, j \in I$, (i) if $i = \langle \mathsf{store}\ a \rangle$ and $j \in \{\langle \mathsf{load}\ a' \rangle, \langle \mathsf{store}\ a' \rangle, \langle \mathsf{cas}\ a' \rangle\}$ such that $a \neq a'$, then $M_{pso}(i, j) = Y$, (ii) if $i \in \{\langle \mathsf{store}\ a \rangle, \langle \mathsf{cas}\ a \rangle\}$ and $j = \langle \mathsf{load}\ a \rangle$, then $M_{pso}(i, j) = E$, (iii) else $M_{pso}(i, j) = N$.

4. Relaxed memory order (RMO) relaxes the order of instructions even more than PSO, by allowing to reorder loads to different addresses too. RMO [20] is specified by M_{rmo}, such that for all memory instructions $i, j \in I$, (i) if $i \in \{\langle \mathsf{load}\ a \rangle, \langle \mathsf{store}\ a \rangle, \langle \mathsf{cas}\ a \rangle\}$ and $j \in \{\langle \mathsf{load}\ a' \rangle, \langle \mathsf{store}\ a' \rangle, \langle \mathsf{cas}\ a' \rangle\}$ such that $a \neq a'$, then $M_{rmo}(i, j) = Y$, (ii) if $i \in \{\langle \mathsf{store}\ a \rangle, \langle \mathsf{cas}\ a \rangle\}$ and $j = \langle \mathsf{load}\ a \rangle$, then $M_{rmo}(i, j) = E$, (iii) else $M_{rmo}(i, j) = N$.

Example. Figure 2 illustrates a concurrent program with two threads that distinguishes between the different memory models in terms of the possible outcomes. Outcome O_1 is allowed by M_{sc}, while other outcomes are not. Outcomes O_1 and O_2 are allowed by M_{tso}. Outcomes O_1, O_2, and O_3 are allowed by M_{pso}. All outcomes O_1, O_2, O_3, and O_4 are allowed by M_{rmo}.

Transactional programs. Let V be a set of *transactional variables*. Let T be a set of *threads*. Let the set C of *commands* be $(\{\mathsf{read}, \mathsf{write}\} \times V) \cup \{\mathsf{xend}\}$. These commands

Initially : $x_1 = y_1 = x_2 = y_2 = 0$	
Thread 1	Thread 2
$x_1 := 1$ $y_1 := 1$ $r_1 := y_2$ $r_3 := x_2$ $x_1 := 2$	$x_2 := 1$ $y_2 := 1$ $r_2 := y_1$ $r_4 := x_1$ $x_2 := 2$

$O_1 : r_1 = 1, r_2 = 1, r_3 = 1, r_4 = 1$
$O_2 : r_1 = 0, r_2 = 0, r_3 = 0, r_4 = 0$
$O_3 : r_1 = 1, r_2 = 1, r_3 = 0, r_4 = 0$
$O_4 : r_1 = 1, r_2 = 1, r_3 = 2, r_4 = 2$

Fig. 2. A concurrent program with some possible outcomes on different memory models

correspond to a read or write of a transactional variable, and to a transaction end. Depending upon the underlying TM, the execution of these commands may correspond to a sequence of hardware memory instructions. For example, a read of a transactional variable may require to check the consistency of the variable by first reading a version number. Similarly, a transaction end may require to copy many variables from a thread-local buffer to global memory. Moreover, the semantics of the write and the xend commands depend on the underlying STM. For example, a (write, v) command does not alter the value of v in a deferred-update STM, whereas it does in a direct-update STM.

We restrict ourselves to purely transactional code, that is, every operation is part of some transaction. We consider transactional programs as our basic sequential unit of computation. We express transactional programs as infinite binary trees on commands, which makes the representation independent of specific control flow statements, such as exceptions for handling aborts of transactions. For every command of a thread, we define two successor commands, one if the command is successfully executed, and another if the command fails due to an abort of the transaction. Note that this definition allows us to capture different retry mechanisms of TMs, e.g., retry the same transaction until it succeeds, or try another transaction after an abort. We use a set of transactional programs to define a multithreaded transactional program. A *transactional program* θ on V is an infinite binary tree $\theta : \mathbb{B}^* \rightarrow C$. A *multithreaded transactional program* $prog = \langle \theta^1 \dots \theta^n \rangle$ on V is a tuple of transactional programs on V. Let *Progs* be the set of all multithreaded transactional programs.

STM correctness. An STM is characterized by the set of histories (sequences of memory instructions) the STM produces for a given transactional program. In order to reason about the correctness of STMs, the history must contain, apart from the sequence of memory instructions that capture the loads and stores to transactional variables in the program, the following information: (i) when transactions finish (captured with commit and abort instructions), (ii) when read and write commands finish (captured with rfin and wfin instructions), and (iii) rollback of stores to transactional variables in V (captured with rollback a). We define $\hat{I} = I_V \cup (\mathsf{rollback} \times V) \cup \{\mathsf{rfin}, \mathsf{wfin}, \mathsf{commit}, \mathsf{abort}\}$, where $I_V \subseteq I$ is the set of memory instructions to the transactional variables V.

Let $O = \hat{I} \times T$ be the set of *operations*. A *history* $h \in O^*$ is a finite sequence of operations. An STM takes as input a transactional program and, depending upon the memory model, produces a set of histories. Formally, a *software transactional memory* is a function $\Gamma : Progs \times \mathbf{M} \rightarrow 2^{O^*}$.

A *correctness property* π is a subset of O^*. It is natural to require that an STM is correct for *all* programs on a *specific* memory model. This is because an STM may be optimized for performance for a specific memory model, while it could be incorrect on weaker models. That is, different implementation versions may be designed for different memory models. An STM Γ is *correct* for a property π under a memory model M if for all programs $prog \in Progs$, we have $\Gamma(prog, M) \subseteq \pi$.

Opacity. We consider opacity [8] as the correctness (safety) requirement of transactional memories. Opacity builds upon the property of strict serializability [17], which requires that the order of conflicting operations from committing transactions is preserved, and the order of non-overlapping transactions is preserved. Opacity, in addition to strict serializability, requires that even aborting transactions do not read inconsistent values. The motivation behind the stricter requirement for aborting transactions in opacity is that in STMs, inconsistent reads may have unexpected side effects, like infinite loops, or array bound violations. Most of the STMs [4,10,18] in the literature are designed to satisfy opacity. However, there do exist STMs that ensure just strict serializability (for example, a variant to McRT STM), and use exception handling to deal with inconsistent reads.

Given a history $h \in O^*$, we define the *thread projection* $h|_t$ of h on thread $t \in T$ as the subsequence of h consisting of all operations op in h such that $op \in \hat{I} \times \{t\}$. Given a thread projection $h|_t = op_0 \ldots op_m$ of a history h on thread t, an operation op_i is *finishing in* $h|_t$ if op_i is a commit or an abort. An operation op_i is *initiating in* $h|_t$ if op_i is the first operation in $h|_t$, or the previous operation op_{i-1} is a finishing statement. Given a thread projection $h|_t$ of a history h on thread t, a consecutive subsequence $x = op_0 \ldots op_m$ of $h|_t$ is a *transaction* of thread t in h if (i) op_0 is initiating in $h|_t$, and (ii) op_m is either finishing in $h|_t$, or op_m is the last operation in $h|_t$, and (iii) no other operation in x is finishing in $h|_t$. The transaction x is *committing* in h if op_m is a commit. The transaction x is *aborting* in h if op_m is an abort. Otherwise, the transaction x is *unfinished* in h. We say that a load of a transaction variable by thread t is *used* in a history h if the load is immediately succeeded by an rfin statement in $h|_t$. Given a history h, we define $usedloads(h)$ as the longest subsequence of h such that all loads of transaction variables in $usedloads(h)$ are used. Given a history h and two transactions x and y in h (possibly of different threads), we say that x *precedes* y in h, written as $x <_h y$, if the last operation of x occurs before the first operation of y in h. A history h is *sequential* if for every pair x, y of transactions in h, either $x <_h y$ or $y <_h x$. An operation op_1 of transaction x and an operation op_2 of transaction y (where x is different from y) *conflict* in a history h if (i) op_1 is a load, store, or a rollback instruction to some transactional variable v and (ii) op_2 is a store or rollback instruction to v.

A history $h = op_0 \ldots op_m$ is *strictly equivalent* to a history h' if (i) for every thread $t \in T$, we have $h|_t = h'|_t$, and (ii) for every pair op_i, op_j of operations in h, if op_i and op_j conflict and $i < j$, then op_i occurs before op_j in h', and (iii) for every pair x, y of transactions in h, where x is a finished transaction, if $x <_h y$, then it is not the case that $y <_{h'} x$. We define *opacity* as the set of histories h such that there exists a sequential history h', where h' is strictly equivalent to $usedloads(h)$. We specify correctness properties using transition systems called TM specifications [7].

$l ::= lv \mid la[idx] \qquad g ::= gv \mid ga[idx] \qquad e ::= f(l, \ldots, l, idx, \ldots, idx) \qquad c ::= f(idx, \ldots, idx)$
$mem_stmt ::= \ g := e \mid l := g \mid l := e \ \mid idx := c \mid l := \mathsf{cas}(g, e, e) \mid \mathsf{rollback}\ g := e$
$tm_stmt ::= \mathsf{rfin} \mid \mathsf{wfin} \mid \mathsf{commit} \mid \mathsf{abort} \qquad fence ::= \mathsf{stfence} \mid \mathsf{ldfence}$
$p ::= \ mem_stmt \ \mid \ tm_stmt \ \mid fence \ \mid \ p\, ;\, p \ \mid \ \textbf{if}\ \ e\ \ \textbf{then}\ p\ \textbf{else}\ p \ \mid \ \textbf{while}\ \ e\ \ \textbf{do}\ p$

Fig. 3. The syntax of RML

TM specifications. A *TM specification* is a 3-tuple $\langle Q, q_{init}, \delta\rangle$, where Q is a set of states, q_{init} is the initial state, and $\delta : Q \times O \to Q$ is a transition function. A history $op_0 \ldots op_m$ is a *run* of the TM specification if there exist states $q_0 \ldots q_{m+1}$ in Q such that $q_0 = q_{init}$ and for all i such that $0 \leq i \leq m$, we have $(q_i, op_i, q_{i+1}) \in \delta$. The *language* L of a TM specification is the set of all runs of the TM specification. A *TM specification* Σ defines a correctness property π if $L(\Sigma) = \pi$. A TM specification for opacity at a coarse-grained alphabet of read, write, commit, and abort statements was developed [7]. To verify the STM algorithms at the low-level atomicity, we build a new TM specification for opacity with the alphabet O. The new TM specification is about 30 times the size of the TM specification for the coarse-grained alphabet, and has about 70,000 states.

3 The RML Language

We introduce a high-level language, RML, to express STM algorithms with hardware-level atomicity on relaxed memory models. The key idea behind the design of RML is to have a semantics parametrized by the underlying memory model. To capture a relaxed memory model, RML defers a statement until the statement is forced to execute due to a fence, and RML reorders or eliminates deferred statements according to the memory model. We describe below the syntax and semantics of RML.

Syntax. To describe STM algorithms in RML, we use local and global integer-valued locations, which are either variables or arrays. We also have a set of array index variables. The syntax of RML is given in Figure 3. A memory statement (denoted by mem_stmt) in RML models an instruction that executes atomically on the hardware. It can, for instance, be a store or a load of a global variable. Moreover, the TM specific statements are denoted by tm_stmt, and fence statements are denoted by $fence$. Let S_M be the set of memory statements, S_{tm} be the set of TM specific statements, and S_F be the set of fence statements in RML. Let P be the set of RML programs.

Semantics. Let G and L be the set of global and local addresses respectively. Let Idx be the set of index variables. Let $q : G \cup L \cup Idx \to \mathbb{N}$ be a *state* of the global and local addresses and the index variables. Let Q be the set of all states. Note that the syntax of RML is defined in a way that the value of an index variable idx may not depend on a global variable. Given a global location g and a valuation q, we write $[\![g]\!]_q \in G$ to denote the global address represented by g in state q. Similarly, we write $[\![l]\!]_q \in L$ to denote the local address represented by a local location l in state q.

Let $\gamma : S_M \times Q \to I \cup \{\mathsf{skip}\}$ be a mapping function for memory statements, which for a given memory statement and a state, gives the generated hardware instruction. For

example, we have $\gamma(g := e, q) = \langle \mathsf{store}\ [\![g]\!]_q \rangle$ in state q, as the statement $g := e$ causes a store to the global address represented by g in state q. The statement rollback $g := e$ is physically a store instruction, as a rollback undoes the effect of a previous store instruction. We define a *local-variables* function $lvars$ such that given an expression e and a state q, $lvars(e, q)$ is the smallest set of local addresses in L such that if the location l appears in e, then the address $[\![l]\!]_q \in lvars(e, q)$. We define a *write-locals* function $lw : S \times Q \rightarrow 2^L$ and a *read-locals* function $lr : S \times Q \rightarrow 2^L$ to obtain the written and read local addresses in a statement respectively. Table 1 gives the formal definitions of the functions γ, lw, and lr. Moreover, we define a mapping function $\gamma_{tm} : S_{tm} \rightarrow S_F \cup \{\mathsf{skip}\}$, which maps the TM specific statements to fence statements. To avoid instructions of two transactions from the same thread to interleave with each other, we define $\gamma_{tm}(\mathsf{commit}) = \gamma_{tm}(\mathsf{abort}) = \mathsf{stfence}$. Moreover, to ensure that during the read of a global variable, the variable is loaded before the read is declared as finished, we define $\gamma_{tm}(\mathsf{rfin}) = \mathsf{ldfence}$. We define $\gamma_{tm}(\mathsf{wfin}) = \mathsf{skip}$.

We now describe when two memory statements can be reordered in a given state under a given memory model. Let $R : S_M \times S_M \times Q \times \mathbf{M} \rightarrow \{true, false\}$ be a *reordering* function such that $R(s_1, s_2, q, M) = true$ if the following conditions hold: (i) $M(\gamma(s_1, q), \gamma(s_2, q)) = Y$, (ii) $lw(s_1, q) \cap lr(s_2, q) = \emptyset$, (iii) $lw(s_1, q) \cap lw(s_2, q) = \emptyset$, and (iv) $lr(s_1, q) \cap lw(s_2, q) = \emptyset$. Here, the first condition restricts reorderings to those allowed by the memory model, and the remaining conditions check for data dependence between the statements. To defer memory statements and execute them in as many ways as possible, we define a model-dependent enqueue function. This function takes as input the current state, the current sequence of deferred statements, a statement to defer, and a memory model, and produces the set of new possible sequences of deferred statements. We define the *enqueue* function $Enq : S_M^* \times S_M \times Q \times \mathbf{M} \rightarrow 2^{S^*}$ such that given a sequence $d = s_1 \ldots s_n$ of memory statements, a statement s, a state q, and a memory model M, $Enq(d, s, q, M)$ is the largest set such that (i) $s_1 \ldots s_k \cdot s \cdot s_{k+1} \ldots s_n \in Enq(d, s, q, M)$ if for all i such that $k < i < n$, we have $R(s_i, s, q, M) = true$, and (ii) if s is of the form $l := g$, then $s_1 \ldots s_k \cdot (l := e) \cdot s_{k+1} \ldots s_n \in Enq(d, s, q, M)$ if for all i with $k < i < n$, we have $R(s_i, s, q, M) = true$, and $M(\gamma(s_k, q), \gamma(s, q)) = E$ where (a) if s_k is $g := f$, then $e = f$, (b) if s_k is $m := g$ or $m := \mathsf{cas}(g, e_1, e_2)$, then $e = m$. Note that the definition of the reordering function restricts the reordering of control and data-dependent statements. Thus, we cannot capture memory models like Alpha, which allows to reorder data-dependent loads. Similarly, the enqueue function restricts the elimination of only load instructions. While this is sufficient to model many hardware memory models, we cannot capture coalesced stores or redundant store elimination.

Given a program p and a sequence d of deferred statements, we define a predicate $allowDequeue(d, p)$ to be $true$ if (i) p is of the form $\{\mathbf{while}\ e\ \mathbf{do}\ p_1 \mid p_1 \in P\}$ or $\{\mathbf{if}\ e\ \mathbf{then}\ p_1\ \mathbf{else}\ p_2 \mid p_1, p_2 \in P\}$, and there exists a memory statement s in d such that $lw(s, q) \cap lvars(e, q) \neq \emptyset$, or (ii) p is a store fence and there exists a statement s of the form $g := l$ in d, or (iii) p is a load fence and there exists a statement s of the form $l := g$ in d. Figure 4 describes the operational semantics of a program in RML parametrized by the underlying memory model.

Table 1. Formal definitions of the functions γ, lw, and lr for a statement s in a state q

Statement s	$\gamma(s,q)$	$lw(s,q)$	$lr(s,q)$
$g := e$	$\langle \mathsf{store}\ [\![g]\!]_q \rangle$	$\emptyset$	$lvars(e,q)$
$l := g$	$\langle \mathsf{load}\ [\![g]\!]_q \rangle$	$\{[\![l]\!]_q\}$	$\emptyset$
$l := e$	skip	$\{[\![l]\!]_q\}$	$lvars(e,q)$
$l := \mathsf{cas}(g,e_1,e_2)$	$\langle \mathsf{cas}\ [\![g]\!]_q \rangle$	$\{[\![l]\!]_q\}$	$lvars(e_1,q) \cup lvars(e_2,q)$
$\mathsf{rollback}\ g := e$	$\langle \mathsf{store}\ [\![g]\!]_q \rangle$	$\emptyset$	$lvars(e,q)$
$idx := c$	skip	$\emptyset$	$\emptyset$

IF TRUE
$$\frac{q[e] \neq 0 \qquad \forall s \in d \cdot lw(s,q) \cap lvars(e,q) = \emptyset}{\langle \mathbf{if}\ e\ \mathbf{then}\ p_1\ \mathbf{else}\ p_2; p,\ q,\ d\rangle \xrightarrow{\mathsf{skip}} \langle p_1; p,\ q,\ d\rangle}$$

IF FALSE
$$\frac{q[e] = 0 \quad \forall s \in d \cdot lw(s,q) \cap lvars(e,q) = \emptyset}{\langle \mathbf{if}\ e\ \mathbf{then}\ p_1\ \mathbf{else}\ p_2; p,\ q,\ d\rangle \xrightarrow{\mathsf{skip}} \langle p_2; p,\ q,\ d\rangle}$$

WHILE TRUE
$$\frac{q[e] \neq 0 \qquad \forall s \in d \cdot lw(s,q) \cap lvars(e,q) = \emptyset}{\langle \mathbf{while}\ e\ \mathbf{do}\ p_1; p,\ q,\ d\rangle \xrightarrow{\mathsf{skip}} \langle p_1; \mathbf{while}\ e\ \mathbf{do}\ p; p,\ q,\ d\rangle}$$

WHILE FALSE
$$\frac{q[e] = 0 \quad \forall s \in d \cdot lw(s,q) \cap lvars(e,q) = \emptyset}{\langle \mathbf{while}\ e\ \mathbf{do}\ p_1; p,\ q,\ d\rangle \xrightarrow{\mathsf{skip}} \langle p,\ q,\ d\rangle}$$

INDEX
$$\frac{}{\langle idx := c; p,\ q,\ d\rangle \xrightarrow{\mathsf{skip}} \langle p,\ q[idx/c],\ d\rangle}$$

SKIP
$$\frac{p' = s \text{ where } \gamma_{tm}(s) = \mathsf{skip}}{\langle p'; p,\ q,\ d\rangle \xrightarrow{s} \langle p,\ q,\ d\rangle}$$

STORE FENCE
$$\frac{p' = s \text{ where } s = \mathsf{stfence} \text{ or } \gamma_{tm}(s) = \mathsf{stfence} \quad \nexists s \text{ in } d \text{ such that } s = g := l}{\langle p'; p,\ q,\ d\rangle \xrightarrow{s} \langle p,\ q,\ d\rangle}$$

LOAD FENCE
$$\frac{p' = s \text{ where } s = \mathsf{ldfence} \text{ or } \gamma_{tm}(s) = \mathsf{ldfence} \quad \nexists s \text{ in } d \text{ such that } s = l := g}{\langle p'; p,\ q,\ d\rangle \xrightarrow{s} \langle p,\ q,\ d\rangle}$$

ENQUEUE
$$\frac{d' \in Enq(d,s,q,M) \quad s \in \{g := e, l := g, l := e, l := \mathsf{cas}(g,e_1,e_2), \mathsf{rollback}\ g := e\}}{\langle s; p,\ q,\ d\rangle \xrightarrow{\mathsf{skip}} \langle p,\ q,\ d'\rangle}$$

DEQUEUE LOAD
$$\frac{allowDequeue(d,p_1) = true \quad q[g] = c \qquad d = (l := g) \cdot d'}{\langle p_1; p,\ q,\ d\rangle \xrightarrow{\gamma(l:=g,q)} \langle p_1; p,\ q[l/c],\ d'\rangle}$$

DEQUEUE CAS FAIL
$$\frac{allowDequeue(d,p_1) = true \quad q[g] = c \quad q[e_1] \neq c \quad d = (l := \mathsf{cas}(g,e_1,e_2)) \cdot d'}{\langle p_1; p,\ q,\ d\rangle \xrightarrow{\langle \mathsf{load}\ [\![g]\!]_q\rangle} \langle p_1; p,\ q[l/c],\ d'\rangle}$$

DEQUEUE CAS SUCCESS
$$\frac{allowDequeue(d,p_1) = true \quad q[e_1] = q[g] \quad q[e_2] = c \quad d = (l := \mathsf{cas}(g,e_1,e_2)) \cdot d'}{\langle p_1; p,\ q,\ d\rangle \xrightarrow{\gamma(\mathsf{cas}(g,e_1,e_2),q)} \langle p_1; p,\ q[g/c][l/c],\ d'\rangle}$$

DEQUEUE STORE
$$\frac{allowDequeue(d,p_1) = true \quad q[e] = c \qquad d = (g := e) \cdot d'}{\langle p_1; p,\ q,\ d\rangle \xrightarrow{\gamma(s,q)} \langle p_1; p,\ q[g/c],\ d'\rangle}$$

DEQUEUE LOCAL
$$\frac{allowDequeue(d,p_1) = true \quad q[e] = c \qquad d = (l := e) \cdot d'}{\langle p_1; p,\ q,\ d\rangle \xrightarrow{\mathsf{skip}} \langle p_1; p,\ q[l/c],\ d'\rangle}$$

DEQUEUE ROLLBACK
$$\frac{allowDequeue(d,p_1) = true \quad q[e] = c \qquad d = (\mathsf{rollback}\ g := e) \cdot d'}{\langle p_1; p,\ q,\ d\rangle \xrightarrow{\mathsf{rollback}\ [\![g]\!]_q} \langle p_1; p,\ q[g/c],\ d'\rangle}$$

Fig. 4. The operational semantics of RML

STM algorithms in RML. A state of a thread carries the information of the program currently being executed, the valuation of the local variables, the deferred statements of the thread, and the location of the transactional program. A *thread-local state* z_l^t

of thread t is the tuple $\langle p^t, q_L^t, D^t, loc^t \rangle$, where p^t is the current RML program being executed by thread t, $q_L^t : L \cup Idx \to \mathbb{N}$ is the valuation of the local and index variables of thread t, D^t is the deferred statements of thread t, and $loc^t \in \mathbb{B}^*$ is the location of the transactional program θ^t. A *state* z of an STM algorithm with T threads is given by $\langle q_G, z_l^1 \dots z_l^T \rangle$, where $q_G : G \to \mathbb{N}$ is the valuation of the global variables of the STM algorithm, and z_l^t is the thread-local state of thread t for $1 \leq t \leq T$. An STM algorithm A is a 4-tuple $\langle p_r, p_w, p_e, z_{init} \rangle$, where p_r, p_w, and p_e are RML programs, and z_{init} is the initial state of the STM algorithm. Moreover, we define a function $\alpha : C \to P$ that maps a transactional command to an RML program, such that $\alpha((\mathsf{read}, k)) = (v := k; p_r)$, $\alpha((\mathsf{write}, k)) = (v := k; p_w)$, and $\alpha(\mathsf{xend}) = p_e$.

Language of an STM algorithm. Let a *scheduler* σ on T be a function $\sigma : \mathbb{N} \to T$. Given a scheduler σ, a transactional program *prog*, and a memory model M, a *run* of an STM algorithm A is a sequence $\langle z_0, i_0 \rangle, \dots \langle z_n, i_n \rangle$ such that $z_0 = z_{init}$, and for all j such that $0 \leq j < n$, if $z_j = \langle q_G, z_l^1 \dots z_l^T \rangle$ and $z_{j+1} = \langle q'_G, {z'}_l^1 \dots {z'}_l^T \rangle$, then (i) $\langle p, q_G \cup q_L, D \rangle \xrightarrow{i_{j+1}} \langle p', q'_G \cup q'_L, D' \rangle$ is a step of RML with memory model M, and (ii) for all threads $t \neq \sigma(j)$, we have ${z'}_l^t = z_l^t$, and (iii) for thread $t = \sigma(j)$, we have ${z'}_l^t = \langle p'', q'_L, D' \rangle$, where (a) if $i_{j+1} \in \{\mathsf{rfin}, \mathsf{wfin}, \mathsf{commit}\}$, then $p'' = \alpha(\theta(loc \cdot 1))$, (b) else if $i_{j+1} = \mathsf{abort}$, then $p'' = \alpha(\theta(loc \cdot 0))$, (c) else $p'' = p'$. A run $\langle z_{init}, i_0 \rangle, \dots, \langle z_n, i_n \rangle$ of an STM algorithm A produces a history h such that h is the longest subsequence of operations in $i_1 \dots i_n$. The *language* $L(A, M)$ of an STM algorithm A under a memory model M is the set of all histories h where there exists a multi-threaded transactional program *prog* and a scheduler σ such that h can be produced by A on *prog* with σ under M. An STM algorithm A is safe for property π under a memory model M if every history in the language of A under M is included in π.

We describe an STM algorithm by p_r, p_w, and p_e programs, and a set of global and a set of local variables, along with their initial values. As an example of an STM algorithm expressed in RML, we present the TL2 algorithm. Similar descriptions can be obtained for DSTM and McRT STM. DSTM is an obstruction-free STM that does not use locks for controlling concurrency. McRT STM is a lock-based direct-update STM.

TL2 algorithm in RML. Transactional locking II (TL2) is an STM algorithm, which is highly popular for its good performance. It is a deferred-update STM, and uses locks to ensure safety. Figure 5 shows four RML programs: p_r (read), p_w (write), p_e (end), and p_a (abort). The program p_a can be called from within p_r, p_w, and p_e. We use the notation $own[V]$ to denote that own is an array of size V. The global variables are $own[V]$, $ver[V]$, $g[V]$, and clk. The local variables are $rs[V]$, $ws[V]$, $lver[V]$, $localclk$, c, and l. The index variables are u and v. *self* denotes the thread number of the executing thread.

Structural properties of STMs. As STMs provide a programmer with a flexible programming paradigm, an STM can involve an arbitrary number of concurrent threads and variables. Thus, an STM algorithm may have an unbounded number of states (corresponding to state of every variable for every thread), where every state has an unbounded number of transitions (corresponding to read or write for every variable). A common technique in checking correctness of arbitrarily sized systems lies in exploiting the inherent symmetry of the system [5,9]. Guerraoui et al. [6] presented a set of four

```
01 program p_a :
02 u := 0;
03 while u < V do
04    u := u + 1;
05    if own[u] = self then own[u] := 0;
06    rs[u] := 0; ws[u] := 0
07 abort

01 program p_r :
02 if localclk = 0 then localclk := clk;
03 if ws[v] = 0 then
04    l := own[v];
05    if l ≠ 0 then p_a
06    l := g[v];
07    lver[v] := ver[v];
08    if localclk ≠ lver[v] then p_a
09    rs[v] := 1;
10 rfin

01 program p_w :
02 ws[v] := 1;
03 wfin

01 program p_e :
02 u := 0;
03 while u < V do
04    u := u + 1;
05    if (ws[u] = 1) then
06       l := cas(own[ws[u], 0, self);
07       if l ≠ self then p_a
08 l := 0;
09 while l < localclk do
10    l := cas(clk, localclk, localclk + 1);
11    localclk := localclk + 1
12 u := 0;
13 while u < V do
14    u := u + 1;
15    if rs[u] = 1 then
16       rs[u] := 0;
17       l := own[u];
18       c := ver[u];
19       if c ≠ lver[u] then p_a
20       if l ≠ 0 then p_a
21 u := 0;
22 while u < V do
23    u := u + 1;
24    if ws[u] = 1 then
25       ver[u] := localclk;
26       g[u] := l;
27 u := 0;
28 while u < V do
29    u := u + 1;
30    if ws[u] = 1 then
31       own[u] := 0;
32       ws[u] := 0;
33 commit
```

Fig. 5. TL2 algorithm in RML

structural properties of STMs for the alphabet of read, write, commit, and abort commands for deferred-update STMs. The authors subsequently verified these properties by hand for different STMs. These structural properties allowed the authors to reduce the problem of verification of an unbounded number of threads and variables to the problem of verification for two threads and two variables. With slight modifications to two of the four properties, the structural properties can be adapted to our framework for both, deferred and direct-update STMs. The properties P2 (*thread symmetry*) and P3 (*variable projection*) are independent of the level of atomicity and the relaxations of the memory model. Thus, these two properties can be directly used in our framework. The property P1, *transactional projection*, originally [6] stated that aborting and pending transactions have no influence on committing transactions, and can thus be projected away. This holds for deferred-update STMs as an aborted transaction does not write to any variable v of the transactional program. In direct-update STMs, an aborting transaction may write a value to a variable, but that value is not read if the transaction aborts. Thus, aborted transactions can still be projected away. On the other hand, pending transactions can be projected away in a coarse grained alphabet [6], but not in our fine-grained

alphabet due to the fact that a pending transaction may be in the process of committing values to memory. Furthermore, we generalize P4, the *monotonicity* property, to handle any number of pending transactions, as opposed to just one pending transaction in the original property [6]. Note that although this generalization is possible for opacity, it cannot be extended to some weaker properties, like strict serializability.

We proved manually that all considered STM algorithms, DSTM, TL2, and McRT STM, satisfy the four structural properties. This allows us to extend the reduction theorem [6] to our framework.

4 The FOIL Tool

We developed a stateful explicit-state model checker, FOIL, that takes as input the RML description of an STM algorithm A, a memory model M, and a correctness property π, and checks whether A is correct for π under the memory model M. FOIL uses the RML semantics with respect to the memory model M to compute the state space of the STM algorithm A, and checks inclusion within the correctness property π. FOIL builds on the fly, the product of the transition system for A and the TM specification for π. If an STM algorithm A is not opaque for a memory model M, FOIL automatically inserts fences within the RML representation of A in order to make A opaque. FOIL succeeds if it is indeed possible to make A opaque solely with the use of fences. In this case, FOIL reports a possible set of missing fences. FOIL fails if inserting fences cannot make A opaque. In this case, FOIL produces a shortest counterexample to opacity under sequential consistency.[1] We implemented FOIL in OCaml. We used FOIL to check the opacity of DSTM, TL2, and McRT STM under different memory models.

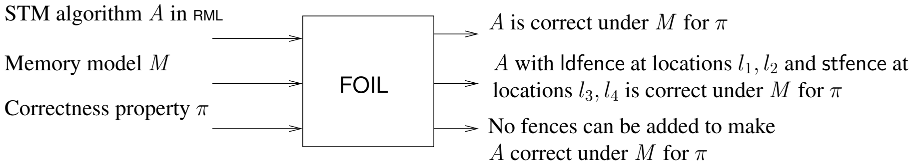

Fig. 6. Inputs and examples of possible outputs of FOIL

Results under sequential consistency. We first model check the STM algorithms for opacity on a sequentially consistent memory model. We find that all of DSTM, TL2, and McRT STM are opaque. The state space obtained for these STM algorithms is large as it covers every possible interleaving, where the level of atomicity is that of the hardware. Table 2 lists the number of states of different STM algorithms with the verification results under sequential consistency. The usefulness of FOIL is demonstrated by the size of state spaces it can handle.

Results under relaxed memory models. Next, we model check the STM algorithms on the following relaxed memory models: TSO, PSO, and RMO. We find that none

[1] Note that if an STM algorithm A cannot be made opaque with fences under some memory model M, then A is not opaque even under sequential consistency.

Table 2. Time for checking the opacity of STM algorithms under sequential consistency on a 2.8 GHz PC with 2 GB RAM. The time is divided into time t_g needed to generate the language of the STM algorithm from the RML description, and time t_i needed to check inclusion within the property of opacity.

STM algorithm A	Number of states	A is opaque?	t_g	t_i
DSTM	1239503	Yes	$212s$	$2.3s$
TL2	2431181	Yes	$471s$	$5.1s$
McRT STM	1756115	Yes	$319s$	$3.9s$

Table 3. Counterexamples generated for opacity, and the type and location of fences required to remove all counterexamples on different relaxed memory models. Instead of the exact location, we list here only the RML program in which the fence has to be introduced.

STM	TSO	PSO	RMO
DSTM	No fences	w_1, stfence: p_e	w_1, stfence: p_e
TL2	No fences	w_1, stfence: p_e	w_1, stfence: p_e w_3, ldfence: p_e w_4, ldfence: p_r
McRT STM	No fences	w_2, stfence: p_a	w_2, stfence: p_a

Counterexamples

w_1 : $(\langle \text{load } v_1\rangle, t_1), (\langle \text{rfin}\rangle, t_1), (\langle \text{store } v_1\rangle, t_2), (\langle \text{store } v_1\rangle, t_1)$
w_2 : $(\langle \text{store } v_1\rangle, t_1), (\langle \text{load } v_2\rangle, t_2), (\langle \text{rfin}\rangle, t_2), (\langle \text{load } v_1\rangle, t_2), (\langle \text{rfin}\rangle, t_2), (\langle \text{rollback } v_1\rangle, t_1)$
w_3 : $(\langle \text{load } v_1\rangle, t_1), (\langle \text{rfin}\rangle, t_1), (\langle \text{load } v_2\rangle, t_2), (\langle \text{rfin}\rangle, t_2), (\langle \text{store } v_1\rangle, t_2), (\langle \text{store } v_2\rangle, t_1)$
w_4 : $(\langle \text{load } v_1\rangle, t_1), (\langle \text{rfin}\rangle, t_1), (\langle \text{store } v_1\rangle, t_2), (\langle \text{load } v_1\rangle, t_1), (\langle \text{rfin}\rangle, t_1)$

of the STM algorithms is opaque for PSO and RMO. FOIL gives counterexamples to opacity. We let FOIL insert fences automatically until the STM algorithms are opaque under different memory models. Table 3 lists the number and location of fences inserted by FOIL to make the various STM algorithms opaque under various memory models. Note that the counterexamples shown in the table are projected to the loads, stores, and rollbacks of the transactional variables, and rfin instructions. We omit the original long counterexamples (containing for example, a sequence of loads and stores of locks and version numbers) for brevity. We give the exact locations of fences for the RML description of TL2 from Figure 5. FOIL discovers that a store fence is needed after the label 26 of p_e under the memory model PSO. Similarly, for RMO, FOIL finds that three fences are needed: one store fence after label 26 in p_e, and two load fences, one after label 17 in p_e and one after label 06 in p_r.

Currently, STM designers use intuition to place fences, as lack of fences risks correctness, and too many fences hamper performance. As FOIL takes as input a memory model, it makes it easy to customize an STM implementation according to the relaxations allowed by the memory model. Although FOIL is not guaranteed to put the minimal number of fences, we found that FOIL indeed inserts the same fences as those in the official STM implementations.

On the need of fences. We note that reordering a store followed by a load, and reading own write early (due to store buffers) does not create a problem in the STMs we have studied. This is evident from the fact that all STMs are correct under the TSO memory model without any fences. On the other hand, relaxing the order of stores or loads can be disastrous for the correctness of an STM. This is because most STMs use version numbers or locks to control access. For example, a reading thread first checks that the variable is unlocked and then reads the variable. A writing thread first updates the variable and then unlocks it. Reversing the order of writes or reads renders the STM incorrect.

5 Related Work

Cohen et al. [3] model checked STMs applied to programs with a small number of threads and variables. They studied safety properties in situations where transactional code has to interact with non-transactional accesses. Guerraoui et al. [6,7] presented specifications for strict serializability and opacity in STM algorithms and model checked various STMs. All these verification techniques in STMs assumed sequentially consistent execution and the atomicity of STM operations like read, write, and commit. The only work that has looked into relaxed memory models in conjunction with transactional memories focused on the testing of TM implementations [15]. We believe that, as with any other concurrent program, it is difficult to eliminate subtle bugs in STM implementations solely with testing.

Our RML language was inspired by +CAL [13], a language for writing and model checking concurrent algorithms. +CAL assumes sequentially consistent behavior, and the notion of an atomic step does not coincide with a hardware atomic step.

There has also been research in guaranteeing sequential consistency under various relaxed memory models [14]. However, conservatively putting fences into STM implementations to guarantee sequential consistency would badly hurt STM performance. STM programmers put fences only where necessary. Also closely related to our work is the CheckFence tool [2], a verifier for concurrent C programs on relaxed memory models. The tool requires as input a bounded test program (a finite sequence of operations) for a concurrent data type and uses a SAT solver to check the consistency and introduce fences where needed. We use the structural properties of STMs which allow us to consider a maximal program on two threads and two variables in order to generalize the result to all programs with any number of threads and variables. Moreover, we model the correctness problem as a relation between transition systems.

6 Conclusion

This paper contributes to bridging the gap between reasoning about the correctness of STMs as described in the literature, typically in high-level pseudo-code assuming a coarse-grained atomicity and sequential consistency, and the correctness of STM implementations on actual multiprocessors. We first presented a formalism to express STMs and their correctness properties at the hardware level of atomicity under relaxed memory models. The formalism is general and encompasses both deferred-update and direct-

update STM schemes. We illustrated our formalism by specifying common STMs such as DSTM, TL2, and McRT STM; memory models such as total store order (TSO), partial store order (PSO), and relaxed memory order (RMO); and correctness criteria such as opacity. We then presented a tool, FOIL, to automatically check the correctness of STMs under fine-grained hardware atomicity and relaxed memory models. FOIL can automatically insert load and store fences where necessary in the STM algorithm description, in order to make the STMs correct under various relaxed memory models. We plan to extend our work to more complicated software memory models, such as Java [16], which further relax the order of memory instructions.

References

1. Adve, S.V., Gharachorloo, K.: Shared memory consistency models: A tutorial. IEEE Computer 29(12), 66–76 (1996)
2. Burckhardt, S., Alur, R., Martin, M.M.K.: CheckFence: Checking consistency of concurrent data types on relaxed memory models. In: PLDI, pp. 12–21. ACM, New York (2007)
3. Cohen, A., Pnueli, A., Zuck, L.D.: Mechanical verification of transactional memories with non-transactional memory accesses. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 121–134. Springer, Heidelberg (2008)
4. Dice, D., Shalev, O., Shavit, N.: Transactional locking II. In: Dolev, S. (ed.) DISC 2006. LNCS, vol. 4167, pp. 194–208. Springer, Heidelberg (2006)
5. Emerson, E.A., Sistla, A.P.: Symmetry and model checking. In: Formal Methods in System Design, pp. 105–131 (1996)
6. Guerraoui, R., Henzinger, T.A., Jobstmann, B., Singh, V.: Model checking transactional memories. In: PLDI, pp. 372–382. ACM, New York (2008)
7. Guerraoui, R., Henzinger, T.A., Singh, V.: Nondeterminism and completeness in transactional memories. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008. LNCS, vol. 5201, pp. 21–35. Springer, Heidelberg (2008)
8. Guerraoui, R., Kapalka, M.: On the correctness of transactional memory. In: PPoPP, pp. 175–184. ACM, New York (2008)
9. Henzinger, T.A., Qadeer, S., Rajamani, S.K.: Verifying sequential consistency on shared-memory multiprocessor systems. In: Halbwachs, N., Peled, D.A. (eds.) CAV 1999. LNCS, vol. 1633, pp. 301–315. Springer, Heidelberg (1999)
10. Herlihy, M., Luchangco, V., Moir, M., Scherer, W.N.: Software transactional memory for dynamic-sized data structures. In: PODC, pp. 92–101. ACM, New York (2003)
11. Herlihy, M., Moss, J.E.B.: Transactional memory: Architectural support for lock-free data structures. In: ISCA, pp. 289–300. ACM, New York (1993)
12. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans. Computers 28(9), 690–691 (1979)
13. Lamport, L.: The $^{+}$CAL algorithm language. In: Najm, E., Pradat-Peyre, J.-F., Donzeau-Gouge, V.V. (eds.) FORTE 2006. LNCS, vol. 4229, p. 23. Springer, Heidelberg (2006)
14. Lee, J., Padua, D.A.: Hiding relaxed memory consistency with a compiler. IEEE Trans. Computers 50(8), 824–833 (2001)
15. Manovit, C., Hangal, S., Chafi, H., McDonald, A., Kozyrakis, C., Olukotun, K.: Testing implementations of transactional memory. In: PACT, pp. 134–143 (2006)
16. Manson, J., Pugh, W., Adve, S.V.: The Java memory model. In: POPL, pp. 378–391. ACM, New York (2005)
17. Papadimitriou, C.H.: The serializability of concurrent database updates. Journal of the ACM 26(4) (1979)

18. Saha, B., Adl-Tabatabai, A., Hudson, R.L., Minh, C.C., Hertzberg, B.: McRT-STM: A high performance software transactional memory system for a multi-core runtime. In: PPOPP, pp. 187–197. ACM, New York (2006)
19. Shavit, N., Touitou, D.: Software transactional memory. In: PODC, pp. 204–213. ACM, New York (1995)
20. Weaver, D., Germond, T. (eds.): The SPARC Architecture Manual (version 9). Prentice-Hall, Inc., Englewood Cliffs (1994)