Proving Non-opacity

Mohsen Lesani and Jens Palsberg

UCLA, University of California, Los Angeles
{lesani,palsberg}@ucla.edu

Abstract. Guerraoui and Kapalka defined opacity as a safety criterion for transactional memory algorithms in 2008. Researchers have shown how to prove opacity, while little is known about pitfalls that can lead to non-opacity. In this paper, we identify two problems that lead to non-opacity, we present automatic tool support for finding such problems, and we prove an impossibility result. We first show that the well-known TM algorithms DSTM and McRT don't satisfy opacity. DSTM suffers from a write-skew anomaly, while McRT suffers from a write-exposure anomaly. We then prove that for direct-update TM algorithms, opacity is incompatible with a liveness criterion called local progress, even for fault-free systems. Our result implies that if TM algorithm designers want both opacity and local progress, they should avoid direct-update algorithms.

1 Introduction

Transactional memory. Atomic statements can simplify concurrent programming that involves shared memory. Transactional memory (TM) [24, 35] interleaves the bodies of atomic statements as much as possible, while guaranteeing noninterleaving semantics. Thus, the noninterleaving in the semantics can coexist with a high degree of parallelism in the implementation. TM aborts an operation that cannot complete without violating the semantics. The use of TM provides atomicity, deadlock freedom, and composability [21], and increases programmer productivity compared to use of locks [30,32]. Researchers have developed formal semantics [1, 26, 29] and a wide variety of implementations of the TM interface in both software [9, 10, 22, 23, 33] and hardware [2, 18]. IBM supports TM in its Blue Gene/Q processor [19], and Intel supports transactional synchronization primitives in its new processor microarchitecture Haswell [7].

Safety. A TM interface consists of the operations `read`, `write`, and `commit`. The task of a TM algorithm is to implement those three operations. What is a correct TM algorithm? The traditional safety criterion for database transactions is strict serializability [31]. For TM algorithms, strict serializability [34] requires that committed transactions together have an equivalent sequential execution, that is, an execution that could also happen if the transactions execute noninterleaved. However, to ensure semantic correctness, active and aborted transactions should execute correctly too. This observation has led researchers to define the stronger safety criteria opacity [13], VWC [25], and TMS1 [11]. We will focus on

opacity, which is the strongest safety criterion and requires that *all* transactions together have an equivalent sequential execution.

Verification. Researchers have shown how to verify the safety of TM algorithms. In pioneering work, Tasiran [36] proved serializability for a class of TM algorithms. Cohen et al. [5,6] were the first to use a model checker to verify strict serializability of TM algorithms for a bounded number of threads and memory locations. Later, Guerraoui and Kapalka [17] proved opacity of two-phase locking with a graph-based approach that is related to an earlier approach to serializability. Guerraoui et al. [14–16] used a model checker to verify opacity of TM algorithms that use an unbounded number of threads and memory locations. Their approach relies on four assumptions about TM algorithms. In follow-up work, Emmi et al. [12] used a theorem prover to generate invariants that are sufficient to prove strict serializability. Their proofs work for TM algorithms that use an unbounded number of threads and memory locations. Later, Lesani et al. [27] presented a TM verification framework based on IO automata and simulation. We identify specific pitfalls that lead to non-opacity and show how a tool can automatically find such pitfalls.

The problem: Which pitfalls lead to non-opacity?

Our results: We identify two problems that lead to non-opacity, we present a tool that automatically finds such problems, we find problems with DSTM and McRT, and we prove an impossibility result.

We show that the well-known TM algorithms DSTM and McRT don't satisfy opacity. These results may be surprising because previous work has proved that DSTM and McRT satisfy opacity [15, 16]. However, there is no conflict and no mystery: the previous work focused on abstractions of DSTM and McRT, while we work with specifications that are much closer to original formulations of DSTM and McRT. Thus, we experience a common phenomenon: once we refine a specification, we may lose some properties.

Let us recall common terminology. A TM algorithm is a *deferred-update* algorithm if every transaction that writes a value must commit before other transactions can read that value. All other TM algorithms are *direct-update* algorithms. DSTM is a deferred-update algorithm while McRT is a direct-update algorithm.

DSTM suffers from a write-skew anomaly, while McRT suffers from a write-exposure anomaly. The write-skew anomaly is an incorrectness pattern that is known in the setting of databases [3]. The write-exposure anomaly happens when a direct-update TM algorithm exposes written values to other transactions before the transaction commits.

We present fixes to both DSTM and McRT that we conjecture make the fixed algorithms satisfy opacity. Interestingly, we note that writers can limit the progress of readers in the fixed McRT algorithm. This is an instance of a general pattern: we prove that for direct-update TM algorithms, opacity is incompatible with a liveness criterion called local progress [4], even for fault-free systems.

Our result implies that if TM algorithm designers want both opacity and local progress, they should avoid direct-update algorithms.

We hope that our observations and tool can help TM algorithm designers to avoid the write-skew and write-exposure pitfalls, and to be aware that if local progress is a goal, then deferred-update algorithms may be the only option.

The rest of the paper. In Section 2 we recall the definition of transaction histories, and in Section 3 we introduce bug patterns that violate opacity. In Section 4, we introduce our tool and in Section 5, we show how our tool automatically finds that DSTM and McRT don't satisfy opacity. In Section 6, we prove that for direct-update TM algorithms, opacity and local progress are incompatible. The full version of the paper has appendices in which we give a formal definition of opacity, prove our theorems, and give details of DSTM, McRT, base objects, and our tool.

2 Histories

Guerraoui and Kapalka [13] defined opacity in terms of *transaction histories*. A transaction history is a record of what happened at the interface of a TM. For example, $H_{WS}, H_{WE}, H_{WE2}, H_1, H_2$ are all transaction histories:

$$\begin{aligned}
H_{WS} = {} & Init \cdot read_{T_1}(1){:}v_0 \cdot read_{T_2}(1){:}v_0 \cdot read_{T_1}(2){:}v_0 \cdot read_{T_2}(2){:}v_0 \cdot \\
& write_{T_1}(1, -v_0) \cdot write_{T_2}(2, -v_0) \cdot \\
& inv_{T_1}(commit_{T_1}) \cdot inv_{T_2}(commit_{T_2}) \cdot ret_{T_1}(\mathbb{C}) \cdot ret_{T_2}(\mathbb{C}) \\
H_{WE} = {} & Init \cdot inv_{T_1}(read_{T_1}(2)) \cdot write_{T_2}(2, v_1) \cdot ret_{T_1}(v_1) \cdot \\
& inv_{T_2}(read_{T_2}(1)) \cdot write_{T_1}(1, v_1) \cdot ret_{T_2}(v_1) \cdot \\
& inv_{T_1}(commit_{T_1}) \cdot inv_{T_2}(commit_{T_2}) \cdot ret_{T_1}(\mathbb{A}) \cdot ret_{T_2}(\mathbb{A}) \\
H_{WE2} = {} & Init \cdot inv_{T_1}((write_{T_1}(1, v_1)) \cdot read_{T_2}(1){:}v_1 \cdot ret_{T_1}(ok) \cdot \\
& write_{T_1}(1, v_2) \cdot commit_{T_1}(){:}\mathbb{C} \cdot commit_{T_2}(){:}\mathbb{A} \\
H_1 = {} & Init \cdot H_0 \cdot write_{T_2}(2, j) \cdot read_{T_1}(2){:}j \cdot write_{T_1}(1, j) \cdot read_{T_2}(1){:}\mathbb{A} \\
H_2 = {} & Init \cdot H_0 \cdot write_{T_2}(2, j) \cdot read_{T_1}(2){:}j \cdot write_{T_1}(1, j) \cdot read_{T_2}(1){:}j
\end{aligned}$$

where $Init$ is described below and H_0 is a transaction history that does not contain a write operation that writes value j.

The invocation event $inv_T(o.n_T(v))$ denotes the invocation of method n on object o in thread T with the argument v. The response event $ret_T(v)$ denotes a response that returns v in the thread T. We will use the term completed method call to denote a sequence of an invocation event followed by the matching response event (with the same thread identifier). We use $o.n_T(v){:}v'$ to denote the completed method call $inv_T(o.n_T(v)) \cdot ret_T(v')$. We use $o.write_T(i, v)$ as an abbreviation for $o.write_T(i, v){:}ok$. Let i range over the set of memory locations, v range over the set of values, and t range over the set of transactions. The interface of a transactional memory object has three methods $read_t(i)$, $write_t(i, v)$ and $commit_t$ and we write calls to those methods without a receiver object. The

current object *this* is the implicit receiver of these calls and thus they are called *this* method calls. The method call $read_t(i)$ returns the value of location i or $\mathbb{A}$ (if the transaction is aborted). The method $write_t(i, v)$ writes v to location i and returns *ok* or returns $\mathbb{A}$. The method $commit_t$ tries to commit transaction t and returns $\mathbb{C}$ (if the transaction is successfully committed) or returns $\mathbb{A}$ (if it is aborted). In general, a transaction history H is of the form $Init \cdot H'$, where $Init$ is the transaction $write_{T_0}(1, v_0), \ldots, write_{T_0}(m, v_0), commit_{T_0}{:}\mathbb{C}$ that initializes every location to v_0, and for all $T \in H'$: $H'|T$ is a prefix of $O.F$ where O is a sequence of reads $read_T(i){:}v$ and writes $write_T(i, v)$ (for some T, i, and v) and F is one of the following sequences: (1) $inv_T(read_T(i))$, $ret_T(\mathbb{A})$ (for some T and i), (2) $inv_T(write_T(i, v))$, $ret_T(\mathbb{A})$ (for some T i, and v), (3) $inv_T(commit_T)$, $ret_T(\mathbb{C})$, or (4) $inv_T(commit_T)$, $ret_T(\mathbb{A})$ (for some T). For a history H, we use $H|T$ to denote the subsequence of all events of T in H. Note that H' is an interleaving of the invocation and response events of different transactions.

3 Opacity and Bug Patterns

Guerraoui and Kapalka [17] defined *final-state opaque* transaction histories. In their earlier, seminal paper on opacity [13], they used the shorter term *opaque* for such histories; we will use opaque and final-state opaque interchangeably. In Appendix A, we formalize opacity as a set of histories called $FinalStateOpaque$ and we prove that none of the transaction histories $H_{WS}, H_{WE}, H_{WE2}, H_1, H_2$ are opaque.

Theorem 1. $\{H_{WS}, H_{WE}, H_{WE2}, H_1, H_2\} \cap FinalStateOpaque = \emptyset$.

We say that H_{WS}, H_{WE}, H_1, H_2 are *bug patterns*, because if a TM can produce any of them, then the TM violates opacity. Let us now focus on H_{WS}, H_{WE} and later turn to H_{WE2}, H_1, H_2.

Write-skew anomaly. The transaction history H_{WS} is evidence of the write-skew anomaly. Let us illustrate the write-skew anomaly with the following narrative.

Assume that a person has two bank accounts that are stored at locations i_1 and i_2 and that have the initial balances v_0 and v_0, where $v_0 > 0$. Assume also that the regulations of the bank require the *sum* of a person's accounts to be positive or zero. Thus, the bank will authorize a transaction that updates the value of one of the accounts with the previous value of the account minus the sum of the two accounts because the transaction makes the sum of the two accounts zero.

Now we interpret the narrative in the context of H_{WS}, which is a record of the execution of two "bank-authorized" transactions. In H_{WS} the transaction T_1 reads the values of both accounts and updates i_1 with $v_0 - (v_0 + v_0) = -v_0$. Similarly, the transaction T_2 reads the values of both accounts and updates i_2 with $-v_0$. But in H_{WS} both transactions commit, which results in a state that violates the regulations of the bank: $-v_0$ is the balance of both accounts.

The problem with H_{WS} stems from that the TM that produced H_{WS} doesn't guarantee noninterleaving semantics of the transactions. In a noninterleaving semantics, either T_1 executes before T_2, or T_2 executes before T_1. However, if we order T_1 before T_2, then the values read by T_2 violate correctness; and if we order T_2 before T_1, then the values read by T_1 violate correctness.

Experts may notice that since H_{WS} is not opaque and all the transactions in H_{WS} are committed, H_{WS} is not even serializable. However, H_{WS} does satisfy *snapshot isolation*, which is a necessary, though not a sufficient, condition for serializability. A history satisfies snapshot isolation if its reads observe a consistent snapshot. Snapshot isolation prevents observing some of the updates of a committing transaction before the commit and some of the rest of the updates after the commit. Algorithms that support only snapshot isolation but not serializability are known to be prone to the write-skew anomaly, as shown by Berenson et al. [3]. Note that H_{WS} satisfies snapshot isolation but suffers from the write-skew anomaly. A TM algorithm that satisfies serializability (and opacity) must both provide snapshot isolation and prevent the write-skew anomaly.

Write-exposure anomaly. The transaction history H_{WE} is evidence of the write-exposure anomaly. The two locations i_1 and i_2 each has initial value v_0 and no *committed* transaction writes a different value to them, and yet the two read operations return the value v_1. Write-exposure happens when a transaction that eventually fails to commit *writes* to a location i and *exposes* the written value to other transactions that read from i. Thus, active or aborting transactions can read inconsistent values. This violates opacity even if these transactions are eventually prevented from committing.

4 Automatic Bug Finding

We present a language called Samand in which a program consists of a TM algorithm, a user program, and an assertion. A Samand program is *correct* if every execution of the user program satisfies the assertion. Our tool solves constraints to decide whether a Samand program is correct. Our approach is reminiscent of bounded model checking: we use concurrency constraints instead of Boolean constraints, and we use an SMT solver instead of a SAT solver.

Our language. We present Samand via two examples. We will use a sugared notation, for simplicity, while in an appendix of the full paper, we list the actual Samand code for both examples. The first example is

$$(\text{Core DSTM}, P_{WS}, \neg WS)$$

where Core DSTM (see Figure 1) is a core version of the TM algorithm DSTM, and the user program and assertion are:

$$\begin{aligned} P_{WS} &= \{read_{T_1}(1){:}r_{11}\ \ read_{T_1}(2){:}r_{12}\ \ write_{T_1}(1,v_1)\ \ commit_{T_1}(){:}c_1\}\ ||\\ &\quad\ \{read_{T_2}(1){:}r_{21}\ \ read_{T_2}(2){:}r_{22}\ \ write_{T_2}(2,v_1)\ \ commit_{T_2}(){:}c_2\}\\ WS &= (r_{11}=v_0\ \wedge\ r_{12}=v_0\ \wedge\ r_{21}=v_0\ \wedge\ r_{22}=v_0\ \wedge\ c_1=\mathbb{C}\ \wedge\ c_2=\mathbb{C}) \end{aligned}$$

Note that the assertion WS specifies a *set* of buggy histories of the user program; the history H_{WS} is a member of that set. The second example is

$$(\text{Core McRT}, P_{WE}, \neg WE)$$

where Core McRT (see Figure 2) is a core version of the TM algorithm McRT, and the user program and assertion are:

$$\begin{aligned} P_{WE} &= \{read_{T_1}(2){:}r_1 \;\; write_{T_1}(1, v_1) \;\; commit_{T_1}(){:}c_1\} \;||\\ &\quad\;\; \{write_{T_2}(2, v_1) \;\; read_{T_2}(1){:}r_2 \;\; commit_{T_2}(){:}c_2\} \\ WE &= (r_1 = v_1 \;\wedge\; r_2 = v_1 \;\wedge\; c_1 = \mathbb{A} \;\wedge\; c_2 = \mathbb{A}) \end{aligned}$$

Like above, the assertion WE specifies a *set* of buggy histories of the user program; the history H_{WE} is a member of that set.

Samand enables specification of loop-free user programs. Every user program has a finite number of possible executions and those executions all terminate.

Each of Core DSTM and Core McRT has three parts: declarations, method definitions, and a program order. Let us take a closer look at these algorithms.

Core DSTM has two shared objects *state* and *start*, and one thread-local object *rset*. Samand supports five types of objects namely `AtomicRegister`, `AtomicCASRegister`, `Lock`, `TryLock`, and `BasicRegister`, as well as arrays and records of such objects. Atomic registers, atomic compare-and-swap (cas) registers, locks, and try-locks are linearizable objects, while basic registers behave as registers only if they are not accessed concurrently. Core DSTM declares one record type *Loc* that has three fields.

Core DSTM has five methods *read*, *write*, *commit*, *stableValue*, and *validate*. Among those, a user program can call the first three, while the *read* method calls the last two, and the *commit* method calls *validate*. Each method is a list of labeled statements that can be method calls on objects, simple arithmetic statements, dynamic memory allocation statements, and if and return statements. The `new` operator dynamically allocates an instance of a record type and returns a reference to it.

Core McRT has three shared objects, two thread-local objects, four methods, and a specification of the program order.

Core McRT specifies the program order $R03 \prec_p R04$, $C03 \prec_p C04$. The idea is to enable out-of-order execution yet maintain fine-grained control of the execution. The execution of the algorithm in a Samand program can be any out-of-order execution that respects the following: the program control dependencies, data dependencies, lock happens-before orders, the declared program orders, that each linearizable object satisfies the linearizability conditions, and that each basic register behaves as a register if it is not accessed concurrently. A method call m_1 is data-dependent on a method call m_2 if an argument of m_1 is the return variable of m_2. If a method call m_2 is data-dependent on a method call m_1 then m_1 must precede m_2 in any execution. For example, in Core McRT, the statement $R03$ must precede $R04$ in any execution. Each statement of the if and else blocks of an if statement is control-dependent on the if statement.

Intuitively, a program execution must respect both the wishes of the programmer and the guarantees of the objects. We can use fences to implement the declared orders.

Constraints. Our tool uses the following notion of constraints to decide whether a Samand program is correct. Let l, x, v range over finite sets of labels, variables, and values, respectively. Let the *execution condition* of a statement be the conjunction of all enclosing if (or else) conditions. A constraint is an assertion about transaction histories and is generated by the following grammar:

$$
\begin{array}{lll}
a ::= obj(l) = o \mid name(l) = n \mid thread(l) = T \mid & & \text{Assertion} \\
\quad arg1(l) = u \mid arg2(l) = u \mid retv(l) = x \mid & & \\
\quad cond(l) = c \mid exec(l) \mid l \prec l \mid \neg a \mid a \wedge a & & \\
u := v \mid x & & \text{Variable or Value} \\
c := u = u \mid u < u \mid \neg c \mid c \wedge c & & \text{Condition}
\end{array}
$$

The assertions $obj(l) = o$, $name(l) = n$, $thread(l) = T$, $arg1(l) = u$, $arg2(l) = u$, $retv(l) = x$ and $cond(l) = c$ respectively assert that the receiver object of l is o, the method name of l is n, the calling thread of l is T, the first argument of l is u, the second argument of l is u, the return value of l is x, and the execution condition of l is c. The assertion $exec(l)$ asserts that l is executed. The assertion $l \prec l'$ asserts that l is executed before l'.

The satisfiability problem is to decide, for a given constraint, whether there exists a transaction history that satisfies the constraint. One can show easily that the satisfiability problem is NP-complete.

From programs to constraints. We map a Samand program to a set of constraints such that the Samand program is correct if and only if the constraints are unsatisfiable.

Let us first define the run-time labels for a program. A run-time label denotes a run-time program point and is either a program label (if the program point is at the top level) or a concatenation of two program labels (if the program point is in a procedure). In the latter case, the additional label is the program label of the caller.

Let us now define the labels and variables that we use in the constraints for a Samand program. For each call we define two labels: the run-time label of the call concatenated with Inv and with Ret, respectively. For other statements we have a single label, namely the run-time label. For each local variable, we define a family of constraint variables, namely one for each caller: each constraint variable is the concatenation of the program label of the caller and the name of the local variable.

Next, we define two auxiliary concepts that are helpful during constraint generation. The *program order* is a total order on program labels. We define the program order to be the transitive closure of the following orders: the control and data dependencies, the declared program order, the orders imposed by locks, that each invocation event is before its matching response event and that each method call inside a *this* method call is before the invocation and after the response event of the *this* method call. The *execution order* is the ordering of labels in a particular history.

We have five sources of constraints: the method calls, the execution conditions, the program order, the base objects, and the assertion.

First, for each run-time label of a method call, we generate constraints that assert the receiver object, the method name, the calling thread, the arguments, the return variable, and the execution condition. For each *this* method call, we generate constraints that assert that the actual parameters and the formal parameters are equal, that the response event of the *this* method call is executed if and only if one (and only one) of its return statements are executed, and that if a return statement is executed, the argument of the return statement is equal to the returned variable of the *this* method call.

Second, we generate constraints that assert that a statement is executed if and only if its execution condition is valid and no prior return statement is executed.

Third, we generate constraints that assert that if l_1 is before l_2 in the program order and the statements with labels l_1, l_2 are both executed, then l_1 is before l_2 in the execution order.

Fourth, we generate constraints that assert the safety properties of the base objects. For each linearizable object, there should be a linearization order of the executed method calls on the object. For example, consider an atomic register. The *write* method call that is linearized last in the set of *write* method calls that are linearized before a *read* method call R is called the *writer* method call for R. The return value of each *read* method call is equal to the argument of its writer method call. For a second example, consider an atomic cas register. A *successful write* is either a *write* method call or a successful *cas* method call. The *written value* of a successful write is its first argument, if it is a *write* method call or is its second argument, if it is a *cas* method call. For a method call m, the successful write method call that is linearized last in the set of successful write method calls that are linearized before m is called the *writer* method call for m. The return value of each *read* method call is equal to the written value of its writer method call. A *cas* method succeeds if and only if its first argument is equal to the written value of its writer method call. For a third example, consider a lock object. The last method call linearized before a *lock* method call is an *unlock* method call. Similarly, the last method call linearized before an *unlock* method call is a lock method call. For a fourth example, consider a try-lock object. We call a *lock* method call or successful *tryLock* method call, a *successful lock* method call. We call a *lock* method call, successful *tryLock* method call or *unlock* method call, a *mutating* method call. The last *mutating* method call linearized before a successful *lock* method call is an *unlock* method call. Similarly, the last *mutating* method call linearized before an *unlock* method call is a *successful lock* method all. A *tryLock* succeeds if the last *mutating* method before it in the linearization order is an *unlock*. It fails otherwise (if the last *mutating* method before it in the linearization order is a *successful lock*). The rules for the return value of *read* method calls are similar to the rule for *tryLock* method calls.

Fifth, we map the assertion in the Samand program to the *negation* of that assertion. As a result, we can use a constraint solver to search for a transaction history that violates the assertion in the Samand program.

Our tool. Our tool maps a Samand program to constraints in SMT2 format and then uses the Z3 SMT solver [8] to solve the constraints. If the constraints are unsatisfiable, then the Samand program is correct. If the constraints are satisfiable, then the Samand program is incorrect and the constraint solver will find a transaction history that violates the assertion in the Samand program. Our tool proceeds to display that transaction history as a program trace in a graphical user interface. Our tool and some examples are available at [28].

5 Experiments

We will now report on running our tool on the two example Samand programs. Our first example concerns Core DSTM.

The context. We believe that Core DSTM matches the *paper* on DSTM [23]. While we prove that Core DSTM doesn't satisfy opacity, we have learned from personal communication with Victor Luchangco, one of the DSTM authors, that the *implementation* of DSTM implements more than what was said in the paper and most likely satisfies opacity.

The bug. DSTM provides snapshot isolation by validating the read set (at $R10$) before the read method returns but fails to prevent write skew anomaly. When we run our tool on $(CoreDSTM, P_{WS}, \neg WS)$, we get an execution trace that matches H_{WS}. Figure 3(a) presents an illustration of the set of DSTM executions that exhibit the bug. Note that this set is a subset of the set of executions that the bug pattern describes. In Figure 3(a), each transaction executes from top to bottom and the horizontal lines denote "barriers", that is, the operations above the line are finished before the operations below the line are started and otherwise the operations may arbitrarily interleave. For example, $read_{T_1}(2){:}v_0$ should finish execution before $write_{T_2}(2, -v_0)$ but $read_{T_1}(1){:}v_0$ and $read_{T_2}(1){:}v_0$ can arbitrarily interleave. In Figure 3(a), T_1 writes to location 1 after T_2 reads from it so T_2 does not abort T_1. T_1 invokes commit and finishes the validation phase ($C01 - C04$) before T_2 effectively commits (executes the *cas* method call at $C05$). The situation is symmetric for transaction T_2. During the validation, the two transactions still see v_0 as the stable value of the two locations; thus, both of them can pass the validation phase. Finally, both of them succeed at *cas*. Note that the counterexample happens when the two commit method calls interleave between $C04$ and $C05$.

The fix. We learned from Victor Luchangco that the *implementation* of DSTM aborts the *writer* transactions of the locations in the read set $rset_T$ during validation of the commit method call. We model this fix by adding the following lines between $C01$ and $C02$ in Core DSTM:

foreach $(i \in dom(rset_t))$ {
  $st := start[i].read();$ $t' := st.writer.read();$ **if** $(t \neq t')$ $state[t'].cas(\mathbb{R}, \mathbb{A})$ }

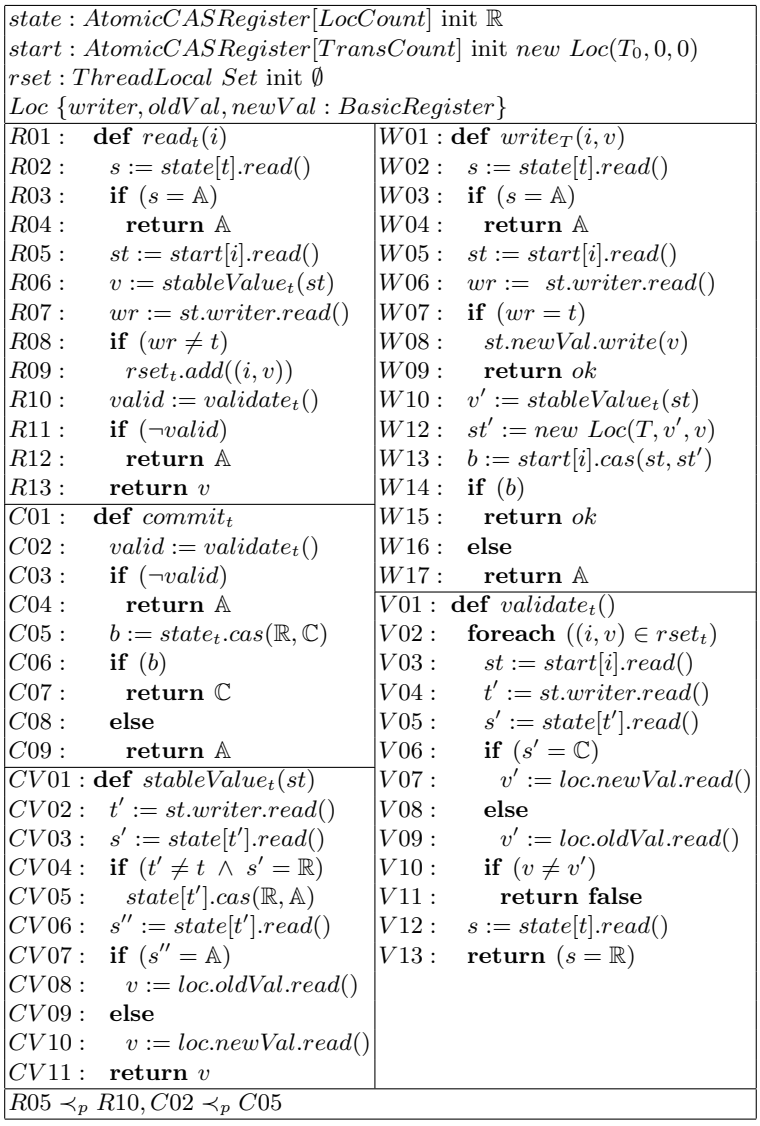

Fig. 1. Core DSTM

Those lines prevent H_{WS} because each transaction will abort the other transaction and thus both of them abort.

Our second example concerns Core McRT.

The context. McRT [33] predates the definition of opacity [13] and wasn't intended to satisfy such a property, as far as we know. Rather, McRT is serializable by design. Still, we prove that Core McRT doesn't satisfy opacity.

The bug. When we run our tool on $(CoreMcRT, P_{WE}, \neg WE)$, we get an execution trace that matches H_{WE} in about 20 minutes. Figure 3(b) presents an

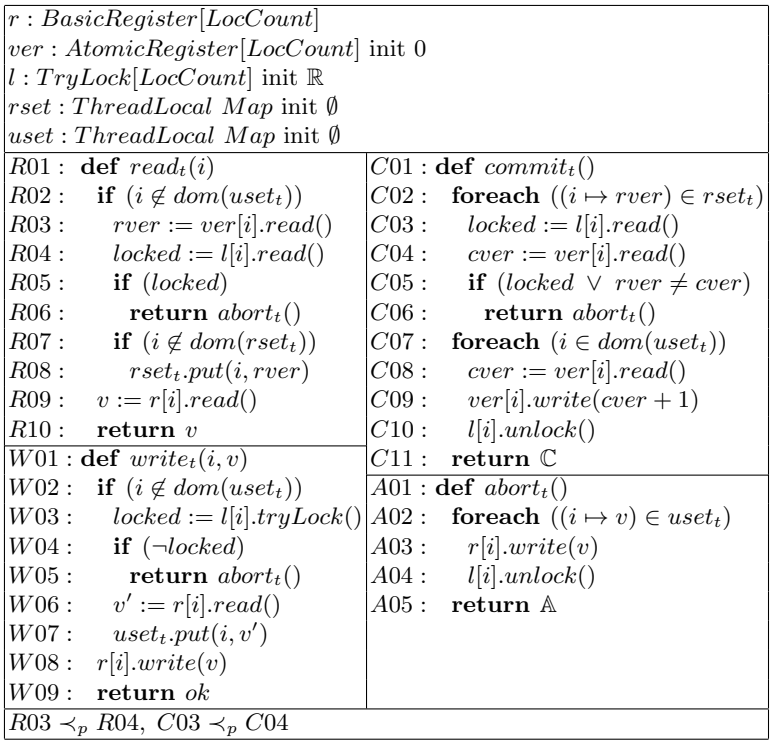

Fig. 2. Core McRT

illustration of the set of executions that exhibit the bug. Like above, this set is a subset of the set of executions that the bug pattern describes. Figure 3(b) uses the same conventions as Figure 3(a). The execution interleaves $write_{T_2}(2, v_1)$ between statements $read_{T_1}(2).R01 - R04$ and $read_{T_1}(2).R05 - R10$ such that the old value of $l[2]$ (unlocked) and the new value of $r[2]$ (the value v_1) are read. Also, $commit_{T_2}.C01 - C04$ are executed before $commit_{T_1}.C05 - C06$ such that T_2 finds $l[1]$ locked and aborts. The situation is symmetric for transaction T_1.

The fix. The validation in the commit method ensures that only transactions that have read consistent values can commit; this is the key to why Core McRT is serializable. Our fix to Core McRT is to let the read method do validation, that is, to insert a copy of lines $C03 - C06$ between line $R09$ and line $R10$ in Core McRT.

Let us use Fixed Core MrRT to denote Core McRT with the above fix. When we run our tool on $(FixedCoreMcRT, P_{WE}, \neg WE)$, our tool determines that the algorithm satisfies the assertion, that is, Fixed Core McRT doesn't have the write-exposure anomaly. The run takes about 10 minutes.

Note though that in the fixed algorithm, a sequence of writer transactions can make a reader transaction abort an arbitrary number of times. This observation motivated the next section's study of progress for direct-update TM algorithms such as McRT.

T_1	T_2
$read_{T_1}(1){:}v_0$ $read_{T_1}(2){:}v_0$	$read_{T_2}(1){:}v_0$ $read_{T_2}(2){:}v_0$
$write_{T_1}(1,-v_0)$ $commit_{T_1}.C01$–$C04$	$write_{T_2}(2,-v_0)$ $commit_{T_2}.C01$–$C04$
$commit_{T_1}.C05$–$C09$	$commit_{T_2}.C05$–$C09$

(a) DSTM counterexamples

T_1	T_2
$read_{T_1}(2).R01$–$R04$	
	$write_{T_2}(2,v_1)$
$read_{T_1}(2).R05$–$R10$	
	$read_{T_2}(1).R01$–$R04$
$write_{T_1}(1,v_1)$	
	$read_{T_2}(1).R05$–$R10$
$commit_{T_1}.C01$–$C04$	$commit_{T_2}.C01$–$C04$
$commit_{T_1}.C05$–$C06$	$commit_{T_2}.C05$–$C06$

(b) McRT counterexamples

Fig. 3. Counterexamples

6 Local Progress and Opacity

We will prove that for direct-update TM algorithms, opacity and local progress are incompatible, even for fault-free systems.

Local progress. We first recall the notion of local progress [4]. Intuitively, a TM algorithm ensures local progress if every transaction that repeatedly tries to commit eventually commits successfully. A *process* is a sequential thread that executes transactions with the same identifier. A process T is *crashing* in an infinite history H if $H|T$ is a finite sequence of operations (not ending in an abort $ret_T(\mathbb{A})$ or commit $ret_T(\mathbb{C})$ response event). A crashing process may acquire a resource and never relinquish it. A process T is *pending* in infinite history H if H has only a finite number of commit response $ret_T(\mathbb{C})$ events. A process *makes progress* in an infinite history, if it is not pending in it. A process T is *parasitic* in the infinite history H if $H|T$ is infinite and in history $H|T$, there are only a finite number of commit invocation $inv_T(commit_T())$ or abort response $ret_T(\mathbb{A})$ events. In other words, a parasitic process is a process that from some point in time keeps executing operations without being aborted and without attempting to commit. A process is *correct* in an infinite history if it is not parasitic and not crashing in the history. A process that is not correct is *faulty*. An infinite history satisfies *local progress*, if every infinite correct process in it makes progress. A TM algorithm ensures *local progress*, if every infinite history of it satisfies local progress and every finite history of it can be extended to an infinite history of it that satisfies local progress. A system is *fault-prone* if at least one process can be crashing or parasitic.

The seminal result. Theorem 2 is the seminal result on the incompatibility of opacity and local progress.

Theorem 2. (Bushkov, Guerraoui, and Kapalka [4]) *For a fault-prone system, no TM algorithm ensures both opacity and local progress.*

Considering a fault-prone system, the proof uses strategies that result in either a crashing or parasitic process.

Fault-prone versus fault-free. The large class of fault-prone systems presents a formidable challenge for designers of TM algorithms who want some form of progress. A crashing or parasitic process may never relinquish the ownership of a resource that another process must acquire before it can make progress. Bushkov, Guerraoui, and Kapalka [4] consider a liveness property called *solo progress* that guarantees that a process that eventually runs alone will make progress. They conjecture that obstruction-free TM algorithms (as defined in [23]) ensure solo progress in parasitic-free systems, and that lock-based TM algorithms ensure solo progress in systems that are both parasitic-free and crash-free. Those conjectures embody the following idea and practical advice.

> **Bushkov, Guerraoui, and Kapalka's advice [4, paraphrased]:** If designers of TM algorithms want opacity and progress, they must consider either weaker progress properties or fault-free systems.

TM algorithms for fault-free systems can rely on that no processes are crashing or parasitic.

Local progress for fault-free systems. Following the advice embodied in the paper by Bushkov, Guerraoui, and Kapalka [4], we study liveness in the setting of fault-free systems. Our main result is that an entire class of TM algorithms cannot ensure both opacity and local progress for fault-free systems.

We need two definitions before we can state our result formally. A TM algorithm is a *deferred-update* algorithm if every transaction that writes a value must commit before other transactions can read that value. All other TM algorithms are *direct-update* algorithms. For example, DSTM is a deferred-update algorithm while McRT is a direct-update algorithm.

Our main result is Theorem 3 which says that direct-update TM algorithms cannot ensure both opacity and local progress for fault-free systems.

Theorem 3. *For a fault-free system, no direct-update TM algorithm ensures both opacity and local progress.*

The proof of Theorem 3 is different from the proof of Theorem 2 because the proof of Theorem 3 cannot use crashing or parasitic processes. The proof of Theorem 3 considers a arbitrary direct-update TM algorithm for a fault-free system and exhibits a particular program that uses the TM. The program leads to transaction histories that are either H_1, H_2, or easily seen to violate local progress. In Theorem 1 we showed that H_1 and H_2 violate opacity.

We can now refine Bushkov, Guerraoui, and Kapalka's advice.

> **Our advice:** If designers of TM algorithms want opacity and *local progress*, they might have success with *deferred-update* TM algorithms that work for fault-free systems.

7 Conclusion

We have identified two problems that lead to non-opacity and we have proved an impossibility result. Our proofs of non-opacity for Core DSTM and Core McRT show that even if an algorithm satisfies opacity at a high level of abstraction, it may fail to satisfy opacity at a lower level of abstraction. Our impossibility result implies that if local progress is a goal, then deferred-update algorithms may be the only option.

Our tool is flexible and can accommodate a variety bug patterns such as H_{WE2} that was suggested by a DISC reviewer (thank you!). Our tool outputs an execution trace of Core McRT that matches H_{WE2} in about 7 minutes. Our tool handles small bug patterns efficiently; scalability is left for future work.

We hope that our observations and tool can help TM algorithm designers to avoid the write-skew, write-exposure, and other pitfalls. We envision a methodology in which TM algorithm designers first use our tool to avoid pitfalls and then use a proof framework such as the one by Lesani et al. [27] to prove correctness. Our tool can be used also during maintenance of TM algorithms. For example, a set of bug patterns can serve as a regression test suite. Additionally, our tool can be used to avoid pitfalls in other synchronization algorithms.

References

1. M. Abadi, A. Birrell, T. Harris, and M. Isard. Semantics of transactional memory and automatic mutual exclusion. In *POPL*, pages 63–74, 2008.
2. C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded transactional memory. In *HPCA*, 2005.
3. Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ANSI SQL isolation levels. *SIGMOD Rec.*, 24(2):1–10, 1995.
4. Victor Bushkov, Rachid Guerraoui, and Michal Kapalka. On the liveness of transactional memory. In *PODC*, pages 9–18, 2012.
5. Ariel Cohen, John W. O'Leary, Amir Pnueli, Mark R. Tuttle, and Lenore D. Zuck. Verifying correctness of transactional memories. In *FMCAD*, 2007.
6. Ariel Cohen, Amir Pnueli, and Lenore D. Zuck. Mechanical verification of transactional memories with non-transactional memory accesses. In *CAV*, 2008.
7. Intel Corporation. Intel architecture instruction set extensions programming reference. 319433-012, 2012.
8. Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient smt solver. In *TACAS*, pages 337–340, 2008.
9. D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *DISC, (LNCS 4167)*, 2006.
10. Dave Dice and Nir Shavit. TLRW: Return of the read-write lock. In *SPAA*, 2010.
11. S. Doherty, L. Groves, V. Luchangco, and M. Moir. Towards formally specifying and verifying transactional memory. *Formal Aspects of Computing*, 2012.
12. Michael Emmi, Rupak Majumdar, and Roman Manevich. Parameterized verification of transactional memories. In *PLDI*, pages 134–145, 2010.
13. R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *PPOPP*, pages 175–184, 2008.

14. Rachid Guerraoui, Thomas A. Henzinger, Barbara Jobstmann, and Vasu Singh. Model checking transactional memories. In *PLDI*, pages 372–382, 2008.
15. Rachid Guerraoui, Thomas A. Henzinger, and Vasu Singh. Software transactional memory on relaxed memory models. In *CAV*, pages 321–336, 2009.
16. Rachid Guerraoui, Thomas A. Henzinger, and Vasu Singh. Model checking transactional memories. *Distributed Computing*, 2010.
17. Rachid Guerraoui and Michal Kapalka. *Principles of Transactional Memory*. Morgan and Claypool Publishers, 2010.
18. L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *ISCA*, 2004.
19. R. Haring, M. Ohnmacht, T. Fox, M. Gschwind, D. Sattereld, K. Sugavanam, P. Coteus, P. Heidelberger, M. Blumrich, R. Wisniewski, A. Gara, G.-T. Chiu, P. Boyle, N. Chist, and C. Kim. The IBM Blue Gene/Q compute chip, 2012.
20. Tim Harris, James Larus, and Ravi Rajwar. *Transactional Memory*. Morgan and Claypool Publishers, second edition, 2010.
21. Tim Harris, Simon Marlow, Simon Peyton, and Jones Maurice Herlihy. Composable memory transactions. In *PPOPP*, pages 48–60. ACM Press, 2005.
22. M. Herlihy, V. Luchangco, and M. Moir. A flexible framework for implementing software transactional memory. In *OOPSLA*, pages 253–262, 2006.
23. M. Herlihy, V. Luchangco, M. Moir, and III W. N. Scherer. Software transactional memory for dynamic-sized data structures. In *PODC*, 2003.
24. Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA*, pages 289–300, 1993.
25. Damien Imbs, José Ramon de Mendivil, and Michel Raynal. Brief announcement: virtual world consistency: a new condition for STM systems. In *PODC*, pages 280–281, 2009.
26. E. Koskinen, M. Parkinson, and M. Herlihy. Coarse-grained transactions. In *POPL*, pages 19–30, 2010.
27. Mohsen Lesani, Victor Luchangco, and Mark Moir. A framework for formally verifying software transactional memory algorithms. In *CONCUR*, 2012.
28. Mohsen Lesani and Jens Palsberg. Proving non-opacity. `http://www.cs.ucla.edu/~lesani/companion/disc13`.
29. K. F. Moore and D. Grossman. High-level small-step operational semantics for transactions. In *POPL*, pages 51–62, 2008.
30. V. Pankratius, A.-R. Adl-Tabatabai, and F. Otto. Does transactional memory keep its promises? results from an empirical study. Technical Report 2009–12, Institute for Program Structures and Data Organization (IPD), University of Karlsruhe, September 2009.
31. Christos H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, 1979.
32. Christopher J. Rossbach, Owen S. Hofmann, and Emmett Witchel. Is transactional programming actually easier? *SIGPLAN Notices*, 45(5), January 2010.
33. Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In *PPoPP*, 2006.
34. M. L. Scott. Sequential specification of transactional memory semantics. In *TRANSACT*, 2006.
35. Nir Shavit and Dan Touitou. Software transactional memory. In *PODC*, 1995.
36. S. Tasiran. A compositional method for verifying software transactional memory implementations. Technical Report MSR-TR-2008-56, Microsoft Research, 2008.

A A Formal Definition of Opacity

This section formalizes the notion of opacity [13] and proves Theorem 1. For completeness, we repeat the definitions in the main body of the paper (about transaction histories).

A.1 Execution Histories

In this section, we define execution histories.

Method calls and events. Let O denote the set of objects, N denote the set method names, $Thread$ denote the set of threads and V denote the set of values. The set of invocation events is $Inv = \{inv_T(o.n_T(v)) \mid o \in O, n \in N, T \in Thread, v \in V\}$. The set of response events is $Res = \{ret_T(v) \mid T \in Thread, v \in V\}$. The set of events is $Ev = Inv \cup Res$. We will use the term completed method call to denote a sequence of an invocation event followed by the matching response event (with the same label). We use $o.n_T(v){:}v'$ to denote the completed method call $inv_T(o.n_T(v)) \cdot ret_T(v')$.

Operations on event sequences. Let E and E' be event sequences. We use $E \cdot E'$ to denote the concatenation of E and E'. For a thread T, we use $E_1 \Subset T_2$ to denote that E_1 is a subsequence of E_2, we use $E|T$ to denote the subsequence of all events of T in E. For an object o, we use $E|o$ to denote the subsequence of all events of o in E. $Sequential$ is the set of sequences of completed method calls possibly followed by an invocation event.

Execution history. An execution history X is a sequence of events where every thread T is sequential (i.e. $X|T \in Sequential$). We assume that each method call in the history has a unique label. The label can be the position of the invocation event of the method call in the history. Let $History$ denote the set of execution histories. Let $Labels(X)$ denote the set of labels in X. As the labels are unique in a history, the following functions on $Labels(X)$ are defined. The functions obj_X, $name_X$, $thread_X$, $arg1_X$, $arg2_X$, $retv_X$ map labels to the receiving object, the method name, the thread identifier, the first and the second argument, and the return value associated with the labels. Similary, iEv and rEv functions on $Labels(X)$ map labels to the invocation and the response events associated with the labels. A history X is equivalent to a history X', $X \equiv X'$, if one is a permutation of the other one, that is, only the events are reordered but the components of the events (including the argument and return values) are preserved. An invocation event e issued by a transaction T is *pending* in an execution history X iff $X|T$ contains no response event that matches and follows e. For an execution history X, we let $Extension(X)$ denote the set of execution histories constructed from X by appending responses for some pending invocations in X. We let $Complete(X)$ denote the subsequence of X that consists of all completed method calls in X.

Real-time relations. Let s be an isogram (i.e. contains no repeating occurrence of the alphabet.) For any $s_1, s_2 \in s$, we write $s_1 \lhd_s s_2$ iff the last element of s_1 occurs before the first element of s_2 in s. For example $ab \lhd_{abcde} de$. For an execution history X, we define the real-time relations $\prec_X$, $\preceq_X$, $\sim_X$, $\precsim_X$

on $Labels(X)$ as follows: First, $l_1 \prec_X l_2$ iff $rEv(l_1) \lhd_X iEv(l_2)$. $l_1 \preceq_X l_2$ iff $l_1 \prec_X l_2 \vee l_1 = l_2$. Second, $l_1 \sim_X l_2$ iff $l_1 \not\prec_X l_2 \wedge l_2 \not\prec_X l_1$. Third, $l_1 \precsim_X l_2$ iff $l_1 \prec_X l_2 \vee l_1 \sim_X l_2$. For an execution history X, we define the thread real-time relations $\lessdot_X$ and $\trianglelefteq_X$ as follows. First, $T \lessdot_X T'$ iff $X|T \lhd_X X|T'$. Second, $T \trianglelefteq_X T'$ iff $T \lessdot_X T' \vee T = T'$.

Objects. For an object o, let $\mathbb{H}(o)$ denote the set of all execution histories that can result from method calls on o. The sequential specification of an object o, denoted by $SeqSpec(o)$, is a set of prefix-closed, sequential histories of o that declares the set of correct sequential histories of o. An object o is basic iff in every sequential execution, it produces a history in its sequential specification i.e. $\mathbb{H}(o) \cap Sequential \subseteq SeqSpec(o)$. In a non-sequential execution history, it may produce an arbitrary history. This is based on the fact that basic objects comply to their sequential specification only if they are accessed sequentially. An object o is linearizable iff every history in $\mathbb{H}(o)$ is linearizable. An execution history X is linearizable for an object o iff there exits histories X' and L such that $X' \in Extension(X|o)$, $L \in Sequential\ L \equiv Complete(X')$, $L \in SeqSpec(o)$, and $\prec_{X|o} \subseteq \prec_L$. We say that such an L is a linearization and $\prec_L$ is a linearization order of $X|o$. In an appendix to the full version of the paper, we specify the classes of objects such as Basic Register, Atomic Register, Atomic CAS Register Strong Counter, Lock and Try-lock.

Shared Memory. The shared memory is a singleton object mem that encapsulates the set of locations Loc where each location loc_i, $i \in I$, $I = \{1, \ldots, m\}$ stores a value $v \in V$. The object mem has three methods $read_t(i)$, $write_t(i, v)$ and $commit_t$. The method call $read_t(i)$ returns the value of loc_i or $\mathbb{A}$ (if the transaction is aborted). The method $write_t(i, v)$ writes v to loc_i and returns ok or returns $\mathbb{A}$. The method $commit_t$ tries to commit transaction T and returns $\mathbb{C}$ (if the transaction is successfully committed) or returns $\mathbb{A}$ (if it is aborted). The object mem is implicit in method calls on mem that is, $read_t(i)$ abbreviates $mem.read_t(i)$.

Transaction History. A transaction history H is $Init \cdot H'$, where $Init$ is the transaction $write_{T_0}(1, v_0)$, $\ldots$, $write_{T_0}(m, v_0)$, $commit_{T_0}$:C that initializes every location to v_0, and for all $T \in H'$: $H'|T$ is a prefix of $O.F$ where O is a sequence of reads $read_T(i)$:v and writes $write_T(i, v)$ (for some $T \in Thread$, $i \in I$, and $v \in V$) and F is one of the following sequences: (1) $inv_T(read_T(i))$, $ret_T(\mathbb{A})$ (for some $T \in Thread$ and $i \in I$), (2) $inv_T(write_T(i, v))$, $ret_T(\mathbb{A})$ (for some $T \in Thread$, $i \in I$, and $v \in V$), (3) $inv_T(commit_T)$, $ret_T(\mathbb{C})$, or (4) $inv_T(commit_T)$, $ret_T(\mathbb{A})$ (for some $T \in Thread$). Let $THistory$ denote the set of transaction histories. The projection of H on i, written $H|i$, denotes the subsequence of history H that contains exactly the events on location i.

A.2 Opacity

A TM algorithm is defined to be opaque if every transaction history of every source program running on top of that TM algorithm is opaque. A history is defined to be opaque if every prefix of it is final-state-opaque. Next, we present the definition of final-state-opacity.

$FinalStateOpaque$ is defined in Figure 4. We use T prefix before some of the terms to avoid confusion with the terms that we defined above for execution histories of objects. We say that a transaction history is sequential if it is a sequence of transactions. A transaction T is committed or aborted in a transaction history H if there is respectively a commit or abort response event for T in H. A completed transaction is either committed or aborted. A live transaction is a transaction that is not completed. A transaction history is complete if all its transactions are completed. A pending transaction has a pending event and a commit-pending transaction has a commit pending event. An extension of a transaction history is obtained by committing or aborting its commit-pending transactions and aborting the other live transactions. If $H \in THistory$ and p is a predicate on transaction identifiers, we define $filter(H, p)$ to be the subsequence of H that contains the events of transactions T for which $p(T)$ is true. The visible history for a transaction T in a sequential transaction history S, $Visible(S, T)$, is the sequence of committed transactions before T in S and T itself. The sequential specification of a location i, $SeqSpec(i)$, is the set of sequential histories of read and write method calls on i where every read returns the value given as the argument to the latest preceding write (regardless of thread identifiers). It is essentially the sequential specification of a register. Transactional sequential specification is the set of complete sequential transaction histories S that for every transaction T and location i, $Visible(S, T)|i$ is in the sequential specification of i. A transaction history H is final-state-opaque if there is an equivalent sequential transaction history S that is real-time-preserving and a member of transactional sequential specification. In other words, every correct concurrent execution is indistinguishable from a correct sequential execution.

Note that opacity and linearizability are at two different levels. In fact, linearizability is the correctness condition for the base concurrent objects. TM algorithms rely on the guarantees of several of these objects to guarantee the correctness conditions for memory transactions.

A.3 Proof of Theorem 1

We now prove Theorem 1, which we restate here.

Theorem 1. $\{H_{WS}, H_{WE}, H_{WE2}, H_1, H_2\} \cap FinalStateOpaque = \emptyset$.

Proof. We consider each of H_{WS}, H_{WE}, H_1, H_2 in turn.

First we consider H_{WS}. We have that

$$\begin{aligned} H_{WS} = Init \cdot\ & read_{T_1}(1){:}v_0 \cdot read_{T_2}(1){:}v_0 \cdot \\ & read_{T_1}(2){:}v_0 \cdot read_{T_2}(2){:}v_0 \cdot \\ & write_{T_1}(1, -v_0) \cdot write_{T_2}(2, -v_0) \cdot \\ & inv_{T_1}(commit_{T_1}) \cdot inv_{T_2}(commit_{T_2}) \cdot ret_{T_1}(\mathbb{C}) \cdot ret_{T_2}(\mathbb{C}) \end{aligned}$$

We will prove the lemma by contradiction. Suppose $H_{WS} \in FinalStateOpaque$. H_{WS} is a complete history, thus $TExtension(H_{WS}) = \{H_{WS}\}$. By definition of

$FinalStateOpaque$, we have that there exists a history S such that (1) $S \in TSequential$, (2) $H_{WS} \equiv S$, (3) $\leq_{H_{WS}} \subseteq \leq_S$ and (4) $S \in TSeqSpec$. From the definition of H_{WS} above, we have that $T_0 \prec_{H_{WS}} T_1$ and $T_0 \prec_{H_{WS}} T_2$. Thus, from [3] we have that (5) $T_0 \prec_S T_1 \ \wedge \ T_0 \prec_S T_2$. From [1], we have that (6) $T_1 \prec_S T_2 \ \vee \ T_2 \prec_S T_1$. From [2], [5] and [6], we have that S is either of the following two histories

- Case $S = H_{WS}|T_0 \cdot H_{WS}|T_1 \cdot H_{WS}|T_2$.
  We have that
  $Visible(S, T_2)|1 =$
  $write_{T_0}(1, v_0)$, $read_{T_1}(1)$:v_0, $write_{T_1}(1, -v_0)$, $read_{T_2}(1)$:v_0
  Thus, $Visible(S, T_2)|1 \notin SeqSpec(1)$. Thus, $S \notin TSeqSpec$, a contradiction to [4].
- Case $S = H_{WS}|T_0 \cdot H_{WS}|T_2 \cdot H_{WS}|T_1$.
  We have that
  $Visible(S, T_1)|2 =$
  $write_{T_0}(2, v_0)$, $read_{T_2}(2)$:v_0, $write_{T_2}(2, -v_0)$, $read_{T_1}(2)$:v_0
  Thus, $Visible(S, T_1)|2 \notin SeqSpec(2)$. Thus, $S \notin TSeqSpec$, a contradiction to [4].

This completes the proof for H_{WS}.

Second we consider H_{WE}. We have that

$$\begin{aligned} H_{WE} = Init \ \cdot \ & inv_{T_1}(read_{T_1}(2)) \cdot write_{T_2}(2, v_1) \cdot ret_{T_1}(v_1) \cdot \\ & inv_{T_2}(read_{T_2}(1)) \cdot write_{T_1}(1, v_1) \cdot ret_{T_2}(v_1) \cdot \\ & inv_{T_1}(commit_{T_1}) \cdot inv_{T_2}(commit_{T_2}) \cdot ret_{T_1}(\mathbb{A}) \cdot ret_{T_2}(\mathbb{A}) \end{aligned}$$

We will prove the lemma by contradiction. Suppose $H_{WE} \in FinalStateOpaque$. H_{WE} is a complete history, thus $TExtension(H_{WE}) = \{H_{WE}\}$. By definition of $FinalStateOpaque$, we have that there exists a history S such that (1) $S \in TSequential$, (2) $H_{WE} \equiv S$, (3) $\leq_{H_{WE}} \subseteq \leq_S$ and (4) $S \in TSeqSpec$. From the definition of H_{WE} above, we have that $T_0 \prec_{H_{WE}} T_1$ and $T_0 \prec_{H_{WE}} T_2$. Thus from [3] we have that (5) $T_0 \prec_S T_1 \ \wedge \ T_0 \prec_S T_2$. From [1], we have that (6) $T_1 \prec_S T_2 \ \vee \ T_2 \prec_S T_1$. From [2], [5] and [6], we have that S is either $H_{WE}|T_0 \cdot H_{WE}|T_1 \cdot H_{WE}|T_2$ or $H_{WE}|T_0 \cdot H_{WE}|T_2 \cdot H_{WE}|T_1$. In both of these cases, we have that $Visible(S, T_1)|2 = write_{T_0}(2, v_0), read_{T_1}(2)$:$v_1$. Thus, as $v_0 \neq v_1$, we have that $Visible(S, T_1)|2 \notin SeqSpec(2)$. Thus, $S \notin TSeqSpec$, a contradiction to [4]. This completes the proof for H_{WE}.

Third, we consider H_{WE2}. We have that

$$\begin{aligned} H_{WE2} = Init \cdot & inv_{T_1}((write_{T1}(1, v_1)) \cdot read_{T_2}(1)\text{:}v_1 \cdot ret_{T_1}(ok) \cdot \\ & write_{T_1}(1, v_2) \cdot commit_{T_1}()\text{:}\mathbb{C} \cdot commit_{T_2}()\text{:}\mathbb{A} \end{aligned}$$

We will prove the lemma by contradiction. Suppose $H_{WE2} \in FinalStateOpaque$. H_{WE2} is a complete history, thus $TExtension(H_{WE2}) = \{H_{WE2}\}$. By definition of $FinalStateOpaque$, we have that there exists a history S such that (1) $S \in TSequential$, (2) $H_{WE2} \equiv S$, (3) $\leq_{H_{WE2}} \subseteq \leq_S$ and (4) $S \in TSeqSpec$. From

the definition of H_{WE2} above, we have that $T_0 \prec_{H_{WE2}} T_1$ and $T_0 \prec_{H_{WE2}} T_2$. Thus from [3] we have that (5) $T_0 \prec_S T_1 \ \wedge \ T_0 \prec_S T_2$. From [1], we have that (6) $T_1 \prec_S T_2 \ \vee \ T_2 \prec_S T_1$. From [2], [5] and [6], we have that S is either $H_{WE2}|T_0 \cdot H_{WE2}|T_1 \cdot H_{WE2}|T_2$ or $H_{WE2}|T_0 \cdot H_{WE2}|T_2 \cdot H_{WE2}|T_1$. In both of these cases, we have that $Visible(S, T_2)|1 = write_{T_0}(1, v_0), read_{T_1}(1){:}v_1$. Thus, as $v_0 \neq v_1$, we have that $Visible(S, T_1)|1 \notin SeqSpec(1)$. Thus, $S \notin TSeqSpec$, a contradiction to [4]. This completes the proof for H_{WE2}.

Fourth, we consider H_1. We have that

$$H_1 = H_0 \cdot write_{T_2}(i_2, j) \cdot read_{T_1}(i_2){:}j \cdot write_{T_1}(i_1, j) \cdot read_{T_2}(i_1){:}\mathbb{A}$$

where H_0 is a history that does not contain a write operation that writes value j.

We prove the lemma based on the following idea. To justify the read of value j by T_1, a committed transaction should have written the value j. The only transaction that writes value j is T_2 but it is aborted.

We will prove the theorem by contradiction. Suppose $H_1 \in FinalStateOpaque$. $TExtension(H_1) = \{H_1', H_1''\}$ where $H_1' = H_1 \cdot commit_{T_1}{:}\mathbb{A}$ and $H_1'' = H_1 \cdot commit_{T_1}{:}\mathbb{C}$. By definition of $FinalStateOpaque$, we have that there exists $H \in TExtension(H_1)$ such that there exists a history S such that (1) $S \in TSequential$, (2) $H \equiv S$, (3) $\leq_H \subseteq \leq_S$ and (4) $S \in TSeqSpec$. We consider two cases:

- Case $H = H_1'$.
  From the definition of H_1' and H_1, we have $\forall T \in H_0 \colon T \prec_{H_1'} T_1 \wedge T \prec_{H_1'} T_2$. Thus, by [3], we have (5) $\forall T \in H_0 \colon T \prec_S T_1 \ \wedge \ T \prec_S T_2$. From [1], we have that (6) $T_1 \prec_S T_2 \ \vee \ T_2 \prec_S T_1$. Thus, from [5], [6] and [2], we have that S is either of the following two histories: $S = S_0 \cdot H_1'|T_2 \cdot H_1'|T_1$ or $S = S_0 \cdot H_1'|T_1 \cdot H_1'|T_2$ where S_0 is a serialization of H_0. For both of these histories, we have that $Visible(S, T_1)|2 = S_0|2 \cdot read_{T_1}(2){:}j$ where no transaction in $S_0|2$ writes value j. Thus, $Visible(S, T_1)|2 \notin SeqSpec(2)$. Thus, $S \notin TSeqSpec$, a contradiction to [4].
- Case $H = H_1''$.
  Similar to the previous case, we will have that $Visible(S, T_1)|2 = S_0|2 \cdot read_{T_1}(2){:}j$.

This completes the proof for H_1.

Fifth, we consider H_2. We have

$$H_2 = H_0 \cdot write_{T_2}(i_2, j) \cdot read_{T_1}(i_2){:}j \cdot write_{T_1}(i_1, j) \cdot read_{T_2}(i_1){:}j$$

where H_0 is a history that does not contain a write operation that writes value j.

We prove the lemma based on the following idea. The two transactions T_1, T_2 have read value j from locations 2 and 1 respectively. The only transaction that writes value j to locations 2 and 1 is T_2 and T_1 respectively. Thus, to justify the two read operations, each of the transactions should have been ordered before the other one in the justifying sequential history, which is impossible.

We will prove the theorem by contradiction. Suppose $H_2 \in FinalStateOpaque$. $TExtension(H_2) = \{H_2', H_2'', H_2''', H_2''''\}$ where
$H_2' = H_2 \cdot commit_{T_1}{:}\mathbb{C} \cdot commit_{T_2}{:}\mathbb{C}$,
$H_2'' = H_2 \cdot commit_{T_1}{:}\mathbb{C} \cdot commit_{T_2}{:}\mathbb{A}$,
$H_2''' = H_2 \cdot commit_{T_1}{:}\mathbb{A} \cdot commit_{T_2}{:}\mathbb{C}$,
$H_2'''' = H_2 \cdot commit_{T_1}{:}\mathbb{A} \cdot commit_{T_2}{:}\mathbb{A}$.
By definition of $FinalStateOpaque$, we have that there exists $H \in TExtension(H_2)$ such that there exists a history S such that (1) $S \in TSequential$, (2) $H \equiv S$, (3) $\leq_H \subseteq \leq_S$ and (4) $S \in TSeqSpec$. We consider four cases:

- Case $H = H_2'$.
  Similar to the proof for H_1, we have that S is either of the following two histories: $S_1 = S_0 \cdot H_2'|T_1 \cdot H_2'|T_2$ or $S_2 = S_0 \cdot H_2'|T_2 \cdot H_2'|T_1$ where S_0 is a serialization of H_0. We consider two cases
  - $S = S_1$:
    We have that $Visible(S, T_1)|2 = S_0|2 \cdot read_{T_1}(2){:}j$. where no transaction in $S_0|2$ writes value j.
    Thus, $Visible(S, T_1)|2 \notin SeqSpec(2)$.
    Thus, $S \notin TSeqSpec$, a contradiction to [4].
  - $S = S_2$:
    We have that $Visible(S, T_2)|1 = S_0|1 \cdot read_{T_2}(1){:}j$. where no transaction in $S_0|1$ writes value j.
    Thus, $Visible(S, T_2)|1 \notin SeqSpec(1)$.
    Thus, $S \notin TSeqSpec$, a contradiction to [4].
- Case $H = H_2''$.
  Similar to the previous case, we will have that $Visible(S, T_1)|2 = S_0|2 \cdot read_{T_1}(2){:}j$.
- Case $H = H_2'''$.
  Similar to the previous case, we will have that $Visible(S, T_2)|1 = S_0|1 \cdot read_{T_2}(1){:}j$.
- Case $H = H_2''''$.
  Similar to the previous case, we will have that $Visible(S, T_2)|1 = S_0|1 \cdot read_{T_2}(1){:}j$.

□

$$
\begin{aligned}
Reads(H) &= \{l_R \mid l_R \in H \ \wedge\ obj_H(l_R) = mem \ \wedge \\
&\qquad name_H(l_R) = read \ \wedge\ retv_H(l_R) \neq \mathbb{A}\} \\
Writes(H) &= \{l_W \mid l_W \in H \ \wedge\ obj_H(l_W) = mem \ \wedge \\
&\qquad name_H(l_W) = write \ \wedge\ retv_H(l_W) \neq \mathbb{A}\} \\
Trans(H) &= \{T \mid \exists l \in H\colon thread_H(l) = T\} \\
TSequential &= \{S \in THistory \mid \preccurlyeq_S \text{ is a total order of } Trans(S)\} \\
Committed(H) &= \{T \mid \exists l \in H\colon thread_H(l) = T \ \wedge\ retv_H(l) = \mathbb{C}\} \\
Aborted(H) &= \{T \mid \exists l \in H\colon thread_H(l) = T \ \wedge\ retv_H(l) = \mathbb{A}\} \\
Completed(H) &= Committed(H) \cup Aborted(H) \\
Live(H) &= Trans(H) \setminus Completed(H) \\
TComplete &= \{H \in THistory \mid \forall T \in Trans(H)\colon T \in Completed(H)\} \\
Pending(H) &= \{T \in Live(H) \mid \exists l \in H\colon thread_H(l) = T \ \wedge \\
&\qquad iEv(l) \Subset H \ \wedge\ \neg(rEv(l) \Subset H)\} \\
CommitPending(H) &= \{T \in Live(H) \mid \exists l \in H\colon thread_H(l) = T \ \wedge\ name_H(l) = commit \\
&\qquad iEv(l) \Subset H \ \wedge\ \neg(rEv(l) \Subset H)\} \\
TExtension(H) &= \{H' \in THistory \mid H \text{ is a prefix of } H' \ \wedge\ \forall T \in H' \Rightarrow T \in H \ \wedge \\
&\qquad Live(H) \setminus CommitPending(H) \subseteq Aborted(H') \ \wedge \\
&\qquad CommitPending(H) \subseteq Completed(H')\} \\
Visible(S,T) &= filter\left(S, \lambda t'.\ (t' = T) \ \vee\ \left((t' \prec_S T) \ \wedge\ t' \in Committed(S)\right)\right) \\
NoWriteBetween_S(l_W, l_R) &= \forall l'_W \in Writes(S)\colon l'_W \preceq_S l_W \ \vee\ l_R \prec_S l'_W \\
SeqSpec(i) &= \{S \in Sequential \mid \forall l_R \in Reads(S)\colon \exists l_W \in Writes(S)\colon \\
&\qquad l_W \prec_S l_R \ \wedge\ NoWriteBetween_S(l_W, l_R) \ \wedge \\
&\qquad retv_S(l_R) = arg1_S(l_W)\} \\
TSeqSpec &= \{S \in TSequential \cap TComplete \mid \forall T \in S\colon \forall i \in I\colon \\
&\qquad (Visible(S,T) \mid i) \in SeqSpec(i)\} \\
FinalStateOpaque &= \{H \in THistory \mid \exists H' \in TExtension(H)\colon \exists S \in TSequential\colon \\
&\qquad H' \equiv S \wedge\ \preccurlyeq_{H'} \subseteq \preccurlyeq_S \ \wedge S \in TSeqSpec\}
\end{aligned}
$$

Fig. 4. *FinalStateOpaque*

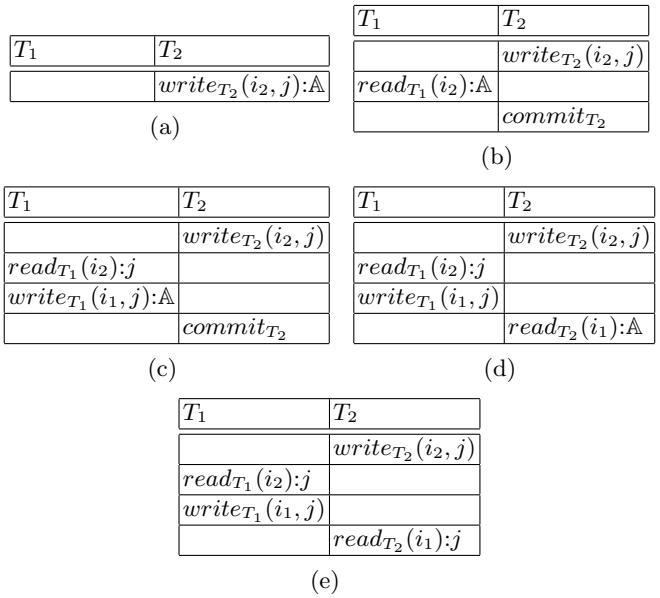

Fig. 5. Impossibility of Opacity and Local-progress for Direct-update TM Algorithms

B Proof of Theorem 3

Proof. Assume otherwise, that is, there is a direct-update algorithm that ensures opacity and local progress. We exhibit a winning strategy for the environment that acts as an adversary to the algorithm and results in either a non-opaque history or an infinite history which does not satisfy local progress. The strategy is as follows. The client iteratively executes the following sequence of operations. Iteration number j is as follows

1. Invoke $write_{T_2}(i_2, j)$.
   If the response is 𝔸,
   leave this iteration and start the next iteration
   Otherwise,
   go to the next step.
2. Invoke $read_{T_1}(i_2)$.
   If the response is 𝔸,
   invoke $commit_{T_2}$ and regardless of the response,
   leave this iteration and start the next iteration.
   Otherwise,
   go to the next step.
3. Invoke $write_{T_1}(i_1, j)$.
   If the response is 𝔸,
   invoke $commit_{T_2}$ and regardless of the response,

leave this iteration and start the next iteration.
Otherwise,
go to the next step.

4. Invoke $read_{T_2}(i_1)$.
Regardless of the response, stop iterating.

In each iteration, the algorithm results in one of the executions depicted in Figure 5. Note that as the algorithm ensures local progress, by definition, every finite history of it can be extended to an infinite history of it. Thus, the three operations of the algorithm should be obstruction-free [4]. Thus, as the operations of the above strategy are called without interleaving from other operations, every operation should return. Also note that as the algorithm is direct-update, the reads return the newly written value j. We consider two cases:

- *The execution stops.* We consider two subcases:
  - The last iteration results in the execution depicted in Figure 5(d), that is, the history is of the form H_1. From Theorem 1 we have that H_1 doesn't satisfy opacity, a contradiction.
  - The last iteration results in the execution depicted in Figure 5(e), that is, the history is of the form H_1. From Theorem 1 we have that H_2 doesn't satisfy opacity, a contradiction.
- *The execution does not stop.* The repeated iterations are depicted in Figure 5(a)-(c). We consider two subcases:
  - From some point in time, only Figure 5(a) is repeated:
    T_2 is executing alone and is aborted an infinite number of times. T_2 has an infinite number of operations, thus is not crashing. T_2 gets an infinite number of abort response events, thus is not parasitic. Therefore T_2 is a correct process. But T_2 does not receive an infinite number of commit response events thus does not make progress. Therefore, the history does not satisfy local progress, a contradiction.
  - Figure 5(b) or (c) happen infinitely often in the history:
    T_1 has an infinite number of operations, thus is not crashing. T_1 gets an infinite number of abort response events, thus is not parasitic. Therefore, T_1 is a correct process. But T_1 does not receive an infinite number of commit response events thus does not make progress. Therefore, the history does not satisfy local progress, a contradiction.

□

C TM Algorithms

C.1 DSTM

DSTM allows only one writer to a location at a time. Therefore, the two fields *oldVal* and *newVal* of Locator are sufficient to represent the values of a location. While the current writer transaction is writing to *newVal*, *oldVal* stores the stable value. Committing a transaction T takes effect by a cas on $state_T$. During the commit, a transaction does not update the *oldVal* of the locations it has written to. Whether *oldVal* or *newVal* is the stable value is decided according to whether the state of the *writer* of the location is *Committed* or *Aborted*. To decide the stable value, if the state of the current *writer* is *Active*, it is *cas*ed to *Aborted*. Then, if the state of *writer* is *Committed*, *newVal* is the stable value; if it is *Aborted*, *oldVal* is the stable value. A new writer transaction of a location needs to set itself as the *writer* and also before overwriting *newVal*, if the state of the previous *writer* is *Committed*, the new writer needs to copy *newVal* (that is the stable value) to *oldVal*. As *writer* and *oldVal* may be concurrently read by readers of the location, the updates of the new writer to them need to be done in isolation. Thus, the first write of a writer transaction instantiates a new Locator object with the current transaction as the *writer*, the stable value of the location as $OldVal$ and the value that it wants to write as *newVal*. Then, the new locator is installed in isolation by a *cas* on $start_i$. On a global read, the pair of the read index and the read value is added to the read set $rset_T$ that is validated before returning from a read or commit method call. Validation checks the equality of the logged value in $rset_T$ to the last committed value of the location and that the transaction is not aborted by others.

The idea. DSTM is a deferred-update TM algorithm (The updates are delayed until a transaction commits [20]). Each location is represented as a reference to a record that stores the last and tentative states of the location. The current value of a location is decided according to the state of the last writer transaction to the location. Thus, a committing transaction updates the value of the locations that it has written to with a single CAS operation on its own state.

The algorithm. Figure 1 shows the Core DSTM algorithm. For a detailed walk-through of the algorithm, see the DSTM paper [23]. Here, we will merely summarize the data structures and then explain a particular execution that proves non-opacity.

The data structures. Let *Loc* be the class of objects with three fields: *writer*, *oldVal* and *newVal*. The field *writer* is a basic register that stores transaction identifiers. The two fields *oldVal* and *newVal* are basic registers that store values. Core DSTM uses the following shared objects. *state* is an array of atomic cas registers of size equal to the number of threads. For each transaction t, $state[t]$ represents the state of t that is $\{\mathbb{R}, \mathbb{A}, \mathbb{C}\}$ (running, aborted or committed) with the default value $\mathbb{R}$. *start* is an array of atomic cas registers of size equal to the number of memory locations. For each memory location i, $start[i]$ stores a reference to a locator object that represents the current state of location

i. The default value is a reference to a locator with *writer* set to T_0. The read set $rset$ is a thread-local (transaction-local) set that stores vectors of index and value of read locations, and is $\emptyset$ initially.

Fix. We learned from Victor Luchangco that the *implementation* of DSTM aborts the *writer* transactions of the locations in the read set $rset_T$ during validation of the commit method call. We can model this fix by adding the following lines between $C01$ and $C02$:

foreach $(i \in dom(rset_T))$
  $st := start[i].read()$
  $t' := st.writer.read()$
  $state[t'].cas(\mathbb{R}, \mathbb{A})$

Those lines prevent H_1 because each transaction will abort the other transaction and thus both of them abort.

Another fix to the algorithm is to let $R09$ store the locator reference (instead of the value) in the read set, and to change the validation for the commit procedure to the following lines:

def $validate_T()$
  foreach $((i, ref) \in rset_T)$
    $start := start[i].read()$
    if $(start \neq ref)$
      return false
  $s := state[T].read()$
  return $s = \mathbb{R}$

Those lines prevent H_1 because both transactions observe that the locator is changed, fail the validation at $C02$ and abort.

C.2 McRT

The first write to location i, tries to acquire l_i at $W03$ before writing to r_i at $W08$. McRT is a direct-update algorithm. It directly writes to r_i during the write method call before the commit method is invoked. Therefore, the old value of r_i is read and cached in the undo set $uset_T$ at $W06-W07$ and restored to r_i while the transaction is aborting at $A02-A04$. A non-local read method call reads ver_i and then l_i at $R03-R04$. If l_i is locked, the transaction is aborted at $R05-R06$. The first non-local read method call from a location caches the version in the read set $rset_T$ at $R08$ which is used during the validation at $C02-C06$. For each read location i, the validation checks that the lock l_i is unlocked and the version ver_i is unchanged since it is read at $R03$. This ensures that r_i is unchanged since it is read at $R09$. For each written location, the version ver_i is incremented at $C08$ and the lock l_i is released at $C09$.

The idea. McRT is a direct-update TM algorithm, which means that transactions directly modify memory locations [20]. Each transaction maintains an undo-log of values that it has overwritten. If the transaction aborts, it restores the old values from the log.

The algorithm. Figure 2 shows the Core McRT algorithm. For a detailed walk-through of the algorithm, see the McRT paper [33]. Here we will merely summarize the data structures and then explain a particular execution that proves non-opacity.

The data structures. Core McRT uses the following shared objects. r is an array of basic registers of size equal to the number of memory locations. For each location i, $r[i]$ stores the value of location i. ver is an array of atomic registers of size equal to the number of memory locations. For each location i, $ver[i]$ stores the version for location i that is initially 0. l is an array of try-locks of size equal to the number of memory locations. For each location i, $l[i]$ is initially released. Core McRT uses the following thread-local (transaction-local) objects. The read set $rset$ is a map from location indices to versions which is $\emptyset$ initially, and the undo set $uset$ is a map from location indices to overwritten values which is $\emptyset$ initially.

In the original implementation, $ver[i]$ and $l[i]$ are stored in a single word. In our specification, we make the distinction explicit and specify the order of accesses to these registers. In addition, the original implementation overwrites the version bits with the transaction descriptor during the lock acquisition. Therefore, the versions had to be cached not only during the read method call but also during the write method call. Our specification stores only versions in the version registers and avoids caching of those registers during the write method call.

Fix to McRT. The validation in the commit method ensures that only transactions that have read consistent values can commit; this is the key to why Core McRT is serializable. A possible fix to make McRT opaque is to let also the read method do validation, that is, to insert a copy of lines $C03 - C06$ between line $R09$ and line $R10$. Note though that in the fixed algorithm, a sequence of writer transactions can make a reader transaction abort an arbitrary number of times. This observation motivated our study of progress for direct-update TM algorithms such as McRT.

D Base Objects

We now define the base objects and their guarantees.

In this subsection, we use $\forall$ and $\exists$ as a notational convenience. $\forall l\colon p$ can be rewritten as $\bigwedge_{(l \in Labels(X))} p$ and $\exists l\colon p$ can be rewritten as $\bigvee_{(l \in Labels(X))} p$.

We first define the abstract types.

Register ADT. A register reg is an object that encapsulates a value and supports $read$ and $write$ methods. The method call $reg.read()$ returns the current encapsulated value of reg. The method call $reg.write(v)$ where $v \in V$ overwrites the encapsulated value of reg with v. For brevity, we write r as a syntactic sugar for $r.read$. Also $r := v$ is used as a syntactic sugar for $r.write(v)$.

The sequential specification of register reg, $SeqSpec(reg)$, is the set of sequential histories of $read$ and $write$ method calls on reg where every read returns the value given as the argument to the latest preceding write (regardless of thread identifiers). (Note that it is assumed that a write method call initializes the register before other methods are invoked.)

$$
\begin{aligned}
isXRead_{X,r}(l_W) &= l_W \in X \ \wedge\ obj_X(l_W) = r \ \wedge\ name_X(l_W) = read \\
isXWrite_{X,r}(l_W) &= l_W \in X \ \wedge\ obj_X(l_W) = r \ \wedge\ name_X(l_W) = write \\
NoWriteBetween_{X,X',r}(l_W, l_R) &= \forall l'_W\colon isXWrite_{X,r}(l'_W) \Rightarrow (l'_W \preceq_{X'} l_W \ \vee\ l_R \preceq_{X'} l'_W) \\
isXWriter_{X,X',r}(l_W, l_R) &= isXWrite_{X,r}(l_W) \ \wedge \\
&\qquad l_W \prec_{X'} l_R \ \wedge \\
&\qquad NoWriteBetween_{X,X',r}(l_W, l_R) \\
SeqSpec(r) &= \{S \in Sequential \mid S|r = S \ \wedge \\
&\qquad \forall l_R\colon isXRead_{S,r}(l_R) \Rightarrow \\
&\qquad \exists l_W\colon isXWriter_{S,S,r}(l_W, l_R) \ \wedge \\
&\qquad\qquad retv_S(l_R) = arg1_S(l_W)\}
\end{aligned}
$$

CAS (Compare-And-Swap) Register ADT. A CAS register is an object that encapsulates a value from the definite set of values V and supports the cas method in addition to $read$ and $write$ methods. The method call $r.cas(v_1, v_2)$ updates the value of the register to v_2 and returns $true$ if the current value of the register is v_1. It returns $false$ otherwise.

We call a $write$ method call or a successful cas method call, a successful write. The sequential specification of cas register reg, $SeqSpec(reg)$, is the set of sequential histories of $read$, $write$ and cas method calls on reg with the following two conditions. Every $read$ returns the value given as the argument to the latest preceding successful write (regardless of thread identifiers). (Note that it is assumed that a write method call initializes the register before other methods are invoked.) Every cas with the first argument v_1 returns $true$ if the latest preceding successful write writes value v_1 and returns $false$ otherwise.

Lock ADT. A lock l is an object that encapsulates an abstract state, acquired $\mathbb{A}$ or released $\mathbb{R}$, and supports the following methods: $lock$: The method call $l.lock$ changes the abstract state from $\mathbb{R}$ to $\mathbb{A}$. $unlock$: The method call $l.unlock$

changes the state from $\mathbb{A}$ to $\mathbb{R}$. *read*: The method call $l.read$ returns *true* if the state of *lock* is $\mathbb{A}$ and $false$ otherwise. The method calls *lock* and *unlock* are mutating method calls. The method call *read* is an accessor method call.

The sequential specification of a lock l, $SeqSpec(l)$, is the set of sequential histories L of *lock*, *unlock*, and *read* method calls on l where the sub-history of L for mutating methods is an alternating sequence of *lock* and *unlock* methods and every *read* method call in L returns *true* if the last mutating method call before it in L is a *lock* and returns $false$ if the last mutating method call before it in L is an *unlock*.

Try-Lock ADT. A try-lock l is an object that encapsulates an abstract state, acquired $\mathbb{A}$ or released $\mathbb{R}$, and supports the following methods: *trylock*: The method call $l.trylock$: if the state of the *lock* is $\mathbb{R}$, it is changed to $\mathbb{A}$ and *true* is returned. Otherwise $false$ is returned. *unlock*: The method call $l.unlock$ changes the state from $\mathbb{A}$ to $\mathbb{R}$. *read*: The method call $l.read$ returns *true* if the state of the *lock* is $\mathbb{A}$ and $false$ otherwise.

We call a lock or successful try-lock, a successful lock method call. We call a lock, successful try-lock or unlock method call, a mutating method call. The sequential specification of a try-lock l, $SeqSpec(l)$, is the set of sequential histories L of *tryLock*, *unlock*, and *read* method calls on l with the following conditions: The last mutating method call before a successful lock method call is an unlock method call. Similarly, the last mutating method call before an unlock method call is a successful lock method call. A *tryLock* method call returns *true* if the latest preceding mutating method call is an *unlock* and returns $false$ otherwise. Similarly, A *read* method call returns *true* if the latest preceding mutating method call is an *unlock* and returns $false$ otherwise.

Counter ADT. A counter c is an object that encapsulates a natural number and supports the following two methods: The method call $c.read$ returns the current value of c. The method call $c.iaf$ increments the value of c and returns the incremented value.

The sequential specification of a counter c, $SeqSpec(c)$, is the set of sequential histories of *read* and *iaf* method calls on c where every method call returns the number of *iaf* method calls before it (including the method call itself). Note that it is assumed that the initial value of the counter is zero.

Set ADT. A set s is an object that represents a set of values and supports the following methods: Add $\oplus$: The method call $s \oplus v$ adds value v to set s. Remove $\ominus$: The method call $s \ominus v$ removes v from s. Membership $\in$: $v \in s$ returns *true* if s contains v and $false$ otherwise. (In fact, $v \in s$ is a syntactic sugar for $s.\ni(v)$.) Iterator: Iterator of elements (used in foreach).

Map ADT. A map m is an object that represents a mapping from a set of keys K to a set of values V and supports the following methods: Add $\oplus$: The method call $m \oplus (k \mapsto v)$ adds or updates the mapping for key $k \in K$ to value $v \in V$ in map m. Remove $\ominus$: The method call $m \ominus k$ removes k from the domain of m. Lookup (): The method call $m(k)$ returns the value that map m associates with key $k \in K$. Domain *dom*: The method call $m.dom$ returns the set of keys that are mapped by map m ($dom(m)$ is a syntactic sugar for $m.dom$). Iterator:

Iterator of key, value pairs (used in foreach).
We have already formally defined basic and linearizable objects. We now define concrete types.

Basic Register. A basic register is a basic object of register ADT.

Let *BasicRegister* denote the class of basic registers.

Atomic Register. An atomic register is a linearizable object of register ADT.

Let *AtomicRegister* denote the class of atomic registers.

Lemma 1. *Consider an atomic register reg. For every read method call R on reg, there is a write method call W on reg that writes the same value that R has returned and W is the last write method call that is linearized before R.*

Formally, if X is a history of an atomic register reg and Reg is the linearization of X, then

$$\begin{aligned}&\forall l_R \colon isXRead_{X,reg}(l_R) \Rightarrow \\ &\exists l_W \colon isXWriter_{X,Reg,reg}(l_W, l_R) \ \wedge \\ &\qquad retv_X(l_R) = arg1_X(l_W)\end{aligned}$$

This is a restatement of Theorem 3 from the original definition of linearizability. Immediate from linearizability of the atomic register and sequential specification of register.

Atomic CAS Register. A CAS register is a linearizable object of CAS register ADT.

Let *AtomicCASRegister* denote the class of Atomic CAS registers.

Lemma 2. *Consider an atomic cas register reg. For every read method call R on reg, there is a successful write W on reg that writes the same value that R has returned and W is the last successful write that is linearized before R.*

Lemma 3. *Consider an atomic cas register reg. A cas method call C on reg with first argument v_1 returns $false$ if the last successful write linearized before C writes v_1 and returns $false$ otherwise.*

Lock. A lock is a linearizable object of lock ADT.

Let *Lock* denote the class of locks.

Intuitively, a history is owner-respecting for a lock if every thread in the history releases the lock only after it has already acquired it. Formally, a history X on a lock l is owner-respecting iff for every thread T, the sub-history of $X|T$ for mutating method calls is a sequence of pairs of *lock* and *unlock* method calls (possibly followed by a *lock* method call).

$$\begin{aligned}&OwnerRespecting_X(lo) = \\ &\quad \forall T, l: \\ &\qquad (l \in X|T \ \wedge \ obj_{X|T}(l) = lo \ \wedge \ name_{X|T}(l) = unlock) \\ &\quad \Rightarrow \exists l': \\ &\qquad (l' \in X|T \ \wedge \ obj_{X|T}(l') = lo \ \wedge \ name_{X|T}(l') = lock \wedge \\ &\qquad l' \prec_{X|T} l \wedge \\ &\qquad \forall l'': (l' \prec_{X|T} l'' \prec_{X|T} l) \Rightarrow \neg(obj_{X|T}(l'') = lo \ \wedge \ name_{X|T}(l'') = unlock)\end{aligned}$$

Lemma 4. *If l is a lock, X is an owner-respecting history of l and L is the linearization of X, then the sub-history of L for mutating method calls is a sequence of pairs of lock and unlock method calls by the same thread (possibly followed by a lock method call).*

This is immediate from the sequential specification of the lock, owner-respecting and real-time-preservation properties.

Lemma 5. *In an owner-respecting execution, if a lock method call of a thread T_1 is linearized before an unlock method call of a thread T_2, then an unlock method call of T_1 is linearized before a lock method call of T_2.*
Formally, if l is a lock, X is an owner-respecting history of l and L is the linearization of X, then

$$
\begin{aligned}
&\forall l_{l1}, l_{u2}: \\
&\quad (l_{l1} \in X \ \wedge \ name_X(l_{l1}) = lock \ \wedge \\
&\quad l_{u2} \in X \ \wedge \ name_X(l_{u2}) = unlock \ \wedge \\
&\quad l_{l1} \prec_L l_{u2}) \Rightarrow \\
&\exists l_{u1}, l_{l2}: \\
&\quad l_{u1} \in X \ \wedge \ name_X(l_{u1}) = unlock \ \wedge \ thread_X(l_{l1}) = thread_X(l_{u1}) \ \wedge \\
&\quad l_{l2} \in X \ \wedge \ name_X(l_{l2}) = lock \ \wedge \ thread_X(l_{l2}) = thread_X(l_{u2}) \ \wedge \\
&\quad l_{u2} \prec_L l_{l1}
\end{aligned}
$$

This is immediate from Lemma 4.

Lemma 6. *In an owner-respecting execution, if a lock method call of a thread T_1 is linearized before a read method call of a thread T_2 that returns $false$, then an unlock method call of T_1 is linearized before the read method call.*
Formally, if l is a lock, X is an owner-respecting history of l and L is the linearization of X, then

$$
\begin{aligned}
&\forall l_{l1}, l_{r2}: \\
&\quad (l_{l1} \in X \ \wedge \ name_X(l_{l1}) = lock \ \wedge \\
&\quad l_{r2} \in X \ \wedge \ name_X(l_{r2}) = read \ \wedge \ retv_X(l_{r2}) = false \\
&\quad l_{l1} \prec_L l_{r2}) \Rightarrow \\
&\exists l_{u1}: \\
&\quad l_{u1} \in X \ \wedge \ name_X(l_{u1}) = unlock \ \wedge \ thread_X(l_{l1}) = thread_X(l_{u1}) \ \wedge \\
&\quad l_{u1} \prec_L l_{r2}
\end{aligned}
$$

This is immediate from Lemma 4 and the sequential specification of the lock.

Try-Lock. A try-lock is a linearizable object of try-lock ADT.
Let $TryLock$ denote the class of try-locks.
Strong Counter. A strong counter is a linearizable object of counter ADT.
Let $SCounter$ denote the class of strong counters.

Lemma 7. *The return values of method calls that are linearized before an iaf method call are smaller than the return value of the iaf method call.*
Formally, if X is a history of a strong counter s and S is the linearization of X, then

$$\begin{aligned}&\forall l, l' : \\ &l \in X \ \wedge \ l' \in X \ \wedge \ name_X(l') = iaf \ \wedge \ l \prec_S l' \Rightarrow \\ &retv_X(l) < retv_X(l')\end{aligned}$$

Immediate from linearizability of the strong counter and sequential specification of counter.

Basic Set. A basic set is a basic object of set ADT.

Let Set denote the class of basic sets.

Basic Map. A basic set is a basic object of map ADT.

Let Map denote the class of basic maps.

E Reconstruction Tool

E.1 A DSL for Concurrent Objects

Separation of specification and implementation is a classical design principle. Unfortunately, this principle is absent in much of the current literature on synchronization algorithms. The algorithms are usually presented in text or in architecture-dependent and optimized code. In particular, the algorithms are tailored for particular memory models and specify particular fences. Striving for efficiency, distinct objects are packed to the bit space of a memory location. To avoid false sharing phenomenon, objects are explicitly padded.

These specifications can have unfortunate effects. Orders that are implicitly respected by a memory model may not be provided by another. Thus, porting algorithms can introduce bugs.[1] This can may require repetition of time-consuming verification efforts. The details of how the layout of an object affects space and time can obfuscate the algorithm intents. So the specifications may not be self-contained, general, portable, amenable to verification and readily understandable.

We note that an algorithm relies on two sets of properties, namely the type of the employed base objects and the preservation of certain orders in the program. A definite type of object such as lock declares the safety and liveness properties that the objects of the type guarantee and is abstract from its implementations. Similarly, the required program orders are abstract from the memory model or the fences needed to satisfy them. We introduce Samand, a DSL where the type of the base objects and the required program orders are explicitly declared. The algorithms that are represented in Samand can be specified and verified once and then translated to multiple low level models.

The specification can specify an assertion as the correctness condition. This assertion can be a partial correctness condition such as negation of a bug pattern. We have built a checking tool that analyzes the specification and verifies the correctness assertion. If the program does not meet the correctness condition, the tool reports an illustrative trace of the program that violates the correctness condition. The history semantics i.e. the set of histories that a specification can generate is a set of constraints. The tool translates the specification into constraints. The negation of the correctness condition is also translated to a constraint. These constraints are represented in the SMT2 format and fed to Z3 SMT solver. If Z3 does not find a model, the specification is verified. If a model is found, an execution that violates the specification assertion is found. The model is read and a program trace is reconstructed from it.

E.2 Example: Dekker Mutual Exclusion

We introduce a DSL called Samand for the specification of concurrent object algorithms. A specification of a concurrent object declares the type of a set of

[1] For example, an earlier release of the STAMP benchmarks had an incorrect port of the TL2 algorithm from SPARC to x86.

```
DekkerSpec {
   f_1: AtomicRegister
   f_2: AtomicRegister
   r: BasicRegister

   def this() {
      W_01>   f_1.write(0)
      W_02>   f_2.write(0)
   }

   main {
      {
         W_1>   f_1.write(1)
         R_2>   x_2 = f_2.read()
         I_1>   if (x_2 = 0)
         C_1>      r.write(1)
      } || {
         W_2>   f_2.write(1)
         R_1>   x_1 = f_1.read()
         I_2>   if (x_1 = 0)
         C_2>      r.write(2)
      }
   }

   order {
      W_1 -> R_2 &&
      W_2 -> R_1
   }

   spec {
      ~ (
         exec(C_1) /\
         exec(C_2)
      )
   }
}
```

Fig. 6. Dekker Algorithm Specification

shared base objects and defines a set of methods. The set of supported base object types are `BasicRegister`, `AtomicRegister`, `AtomicCASRegister`, `Lock` and `TryLock`. There is also support for arrays of these types and thread-local objects. User can define record types. A record type contains a set of object declarations. The `new` operator dynamically allocates an instance of a record type and returns a reference to it. The method definitions call methods on the base objects. Method calls are ordered by program control and data dependencies and lock happens-before orders. To allow for performance benefits of out-of-order

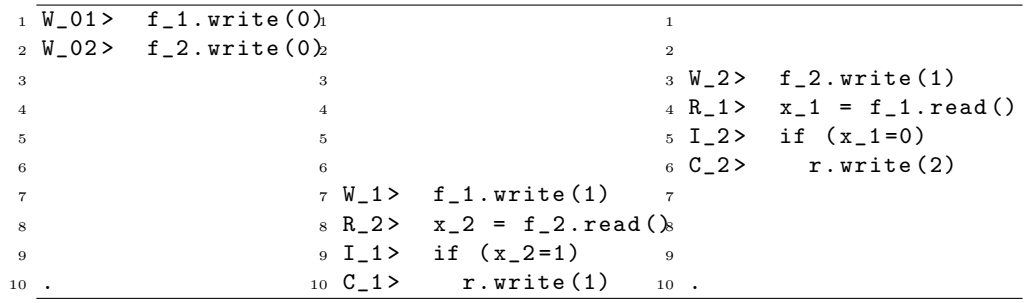

Fig. 7. Bug Trace for Incorrect If Condition

execution, method calls that are unordered by the program are allowed to appear reordered in the histories of the program. The user can explicitly require specific orders in the `order` block. Note that these orders can be translated to fences for specific architectures. In order to represent complete specifications, there is no implicit program order in the language. The language enforces the discipline that the object types and the program order are explicitly declared.

In the `main` block, the user can write a concurrent program that calls the methods of the specified concurrent object. The `main` block is a sequence of blocks, one for each thread. Finally, the `spec` block specifies the correctness assertion. Every history of the concurrent program is expected to satisfy the correctness assertion. The correctness assertion can assert a partial correctness condition. In particular, it can be the negation of a bug pattern.

The set of histories of a specification are constrained by two set of constraints. Firstly, every history respects the guarantees of the base objects. For example, if a base object is an atomic register, then the sub-history for that register should be linearizable. Secondly, every history respects the control, data and program order dependencies. For example, if a method call is data-dependent on another method call, then the latter should precede the former in the history.

Figure 6 shows the specification of Dekker mutual exclusion algorithm in Samand. The two flags are declared as atomic registers. The optional `this` method specifies the initialization statements. This method is executed before the concurrent execution begins. This simple specification does not define any other method. The `main` block specifies the concurrent program. The `order` block specifies that each thread should set its own flag before reading the other thread's flag. Finally, the `spec` block specifies the correctness assertion i.e. the two critical sections should not both execute.

Running Samand checker on the specification of Dekker results in approval of the specification.

If the specification is not met, the Samand checker reports the trace that leads to violation of the specification in a graphical user interface. If the condition of the statement at line `I_1` is replaced with the incorrect condition `(x_2 = 1)`, Samand checker shows the interleaving depicted in Figure 7. If the declared

```
W_02>  f_2.write(0)                                
W_01>  f_1.write(0)                                
                                R_2>  x_2 = f_2.read()
                                I_1>  if (x_2=0)      
                                                      W_2>  f_2.write(1)
                                                      R_1>  x_1 = f_1.read()
                                W_1>  f_1.write(1)    
                                                      I_2>  if (x_1=0)
                                                      C_2>    r.write(2)
.                               C_1>       r.write(1) .
```

Fig. 8. Bug Trace for Removed Program Order

```
W_02>  f_2.write(0)                                
W_01>  f_1.write(0)                                
                                W_1>  f_1.write(1)    
                                R_2>  x_2 = f_2.read()
                                                      W_2>  f_2.write(1)
                                                      R_1>  x_1 = f_1.read()
                                I_1>  if (x_2=0)      
.                               C_1>    r.write(1)    .
```

Fig. 9. Dekker Random Execution

order `W_1 -> R_2` is removed, Samand checker shows the interleaving depicted in Figure 8.

Samand checker is not only a checking tool but can also be viewed as an execution tool. The `false` literal is an assertion that any execution violates. Therefore, declaring `false` as the specification assertion results in a random execution. Updating the spec block of the dekker specification as follows shows an execution instance such as the execution depicted in Figure 9. In this execution only one of the critical sections `C_1` is executed.

```
spec {
   false
}
```

E.3 Language

The set of currently supported object types are basic registers `BasicRegister`, atomic registers `AtomicRegister`, atomic cas registers `AtomicCASRegister`, locks `Lock` and try-locks `TryLock`. As defined in the base objects section, atomic registers, atomic cas registers, locks and try-locks are linearizable objects and basic registers behave as registers only if they are not accessed concurrently.

A base object called `r` of type `BasicRegister` is declared as follows:

```
r: BasicRegister
```

There is also support for arrays. The following declaration declares an array of try-locks objects of size 10.

```
tryLocks: TryLock[10]
```

The 7th element of the array can be accessed by `tryLocks[6]`. There is also support for thread-local objects. A thread-local basic register can be declared as

```
reg: TLocal BasicRegister
```

Thread-local objects are arrays in nature. The thread identifier is implicitly passed for accesses to thread-local variables, and hence thread-local variables are conveniently accessed as normal objects. It is also possible to declare thread-local arrays. User-defined record types are also supported. For example, a `Node` type can be defined as follows:

```
Node {
   lock: Lock
   value: BasicRegister
   next: BasicRegister
}
```

A specification can declare methods. For instance, the following lines show the declaration of a transfer method.

```
def transfer(a) {
   L>     lock.lock()
   R1>    v1 = b1.read()
   R2>    v2 = b2.read()
   C1>    v3 = v1 - a
   C2>    v4 = v2 + a
   W1>    b1.write(v3)
   W2>    b2.write(v4)
   U>     lock.unlock()
   F>     return
}
```

Each method declaration has an implicit parameter for the calling thread identifier. The variable name `t` is reserved for this parameter and should not be used to name any other variable. The argument for this parameter is automatically passed at the call site. A statement is either a method call, a record creation, an if statement, a return statement or a math statement. The following statement allocates memory for an object of record type `Node` and returns a reference to it that is assigned to `ref`.

```
ref = new Node()
```

The following statement calls the method `method` on the base object `object` with the argument `arg` and assigns the return value to `ret`.

```
ret = object.method(arg)
```

If no receiver object is specified for a method call, the receiver is the current object. The following statement calls the method `method` on the field `object` of the record referenced by `ref` with the argument `arg` and assigns the return value to `ret`.

```
ret = ref.object.method(arg)
```

The supported math statements are of the form `x3 = x1 + x2` or `x3 = x1 - x2`.

The `main` block specifies the concurrent program. The thread blocks are separated by ||.

Data and control dependencies order method call. Correctness of the specified algorithm may be dependent on a specific order of method calls that are not ordered by data and control dependencies. The user can declare the required order of method calls in the `order` block. The program order of a specification is the transitive closure of data and control dependencies, the declared orders and the following conventional orders for locks and `this` method calls.

Locks (and try-locks) as the foundation of mainstream language memory models have ordering implications (in addition to the linearizability property). Every statement after a lock method (or a successful try-lock method) is ordered after it and every statement before an unlock method is ordered before it. Method calls on `this` object have ordering implications as well. A function call whose side effects are not clear is even stronger than a compiler barrier. This excludes inline functions and functions known to be pure. We consider full ordering for method calls on `this` object. The statements before and after method calls on `this` object (and their enclosing statements) are ordered respectively before and after the call. In addition, the statements of `this` and `~this` methods are ordered respectively before and after all the statements of the concurrent program (the `main` block).

Finally, the `spec` block specifies the assertion that every history of the specification should satisfy. Note that the correctness assertion can assert complete or partial correctness of the specification such as the negation of a bug pattern. The assertion language supports conjunction `/\`, disjunction `\/`, negation `~` of assertions. Currently, atomic assertions can be that a specific method call is executed

```
exec(M)
```

a method call is executed before another method call

```
M1 \prec M2
```

an equality for variables and values

```
x1 = 2
x1 = x2
```

and `true` and `false` literals.

E.4 Implementation

The specification is analyzed and a skeleton execution graph is built. The constraints are generated from the specification. The constraints are fed to the SMT solver. If it finds a model for the constraints, the model represents the violating execution. The execution order is extracted from the model and added to the execution graph. A topological sort of the graph is computed and the resulting trace is rendered to the user. We describe this process in more detail in the following paragraphs.

We analyze the specification and extract the following information that is later consulted during building the execution graph. We compute the control and data dependencies of the specification. Then, we compute the complete set of dependencies in the specification as the transitive closure of the control and data dependencies, the declared program orders and also the orders imposed by locks and `this` method calls. Each statement of the specification is in the scope of (a possibly empty) sequence of if and else conditions. For each statement, we call the conjunct of these conditions the execution condition of the statement. We compute the execution condition of every base object method call and return statement. For a method definition n and for a method call or return statement l, let $preReturns_n(l)$ be the set of return statements before l in n. We compute $preReturns_n(l)$ for each method definition n and each method call or return statement l in n.

The execution graph is an inlined representation of the concurrent program. Edges between statements of the graph represent execution order. Each method call on a linearizable object and also each return statement is represented as a node in the graph. Each method call on a basic object is represented as two invocation and response nodes in the graph. There is a node for every if and also every else statement and there is an edge from them to any statement in their scope. There are invocation and response nodes for each method call on `this` object. We call the nodes corresponding to the statements of a method call on `this` object the body nodes. The body nodes are inlined between the invocation and response events of the call in the graph. There is an edge from the invocation node to every body node and from every body node to the response node. We use the dependencies between the statements of the specification that we computed before to add dependencies between the nodes of the execution graph. If the statements of two nodes are dependent, an edge is added between the nodes in the execution graph. An edge is added from every node of the `this` method to every node of the concurrent program and from every node of the concurrent program to every node of the `~this` method. Also, using the execution conditions for statements that we computed before, we compute the execution conditions for nodes.

We next generate constraints. We assert the properties of each node in the graph. For example for a method call, we assert the receiver object, the name of the method, the arguments and the return variable. We also assert that a node is executed if and only if its execution condition (computed above) is valid and

none of its pre-returns are executed. We also assert the edges of the graph as the program order.

We represent the following properties of the execution order as constraints. The execution order is a total order (transitive, asymmetric and total) on the set of executed nodes. If there is an edge from a node to another and both are executed, then the former is executed before the latter.

The following constraints are asserted for method calls on `this` object. The parameter variables and arguments are equal. If the response event is executed, one (and only one) of the return nodes are executed and the returned variable is equal to the argument of the return statement. If a return node is executed, the response event is executed and the returned value is equal to the argument of the return statement.

The safety of each base object is known according to its declared type. We represent the safety properties of the base objects as constraints. Locks, try-locks atomic registers and atomic cas registers are linearizable objects. For each linearizable object, the linearization order is the subset of the execution order on the set of executed labels on the object.

Consider an atomic register. The *write* method call that is linearized last in the set of *write* method calls that are linearized before a *read* method call R is called the *writer* method call for R. The return value of each *read* method call is equal to the argument of its writer method call. Consider an atomic cas register. A *successful write* is either a *write* method call or a successful *cas* method call. The *written value* of a successful write is its first argument if it is a *write* method call or is its second argument if it is a *cas* method call. For a method call m, the successful write method call that is linearized last in the set of successful write method calls that are linearized before m is called the *writer* method call for m. The return value of each *read* method call is equal to the written value of its writer method call. A *cas* method succeeds if and only if its first argument is equal to the written value of its writer method call. A basic register behaves similar to an atomic register only if there are no concurrent method calls on it that is for every pair of method calls on it, the response event of one is before the invocation event the other one.

Consider a lock object. The last method call linearized before a *lock* method call is an *unlock* method call. Similarly, the last method call linearized before an *unlock* method call is a lock method call. Consider a try-lock object. We call a *lock* method call or successful *tryLock* method call, a *successful lock* method call. We call a *lock* method call, successful *tryLock* method call or *unlock* method call, a *mutating* method call. We call a failed *tryLock* or *read* method call, an *accessor* method call. The last *mutating* method call linearized before a successful *lock* method call is an *unlock* method call. Similarly, the last *mutating* method call linearized before an *unlock* method call is a *successful lock* method all. A *tryLock* succeeds if the last *mutating* method before it in the linearization order is an *unlock*. It fails otherwise (if the last *mutating* method before it in the linearization order is a *successful lock*). The rules for the *read* method call are similar to the rules for *tryLock* method call.

The negation of the specification assertion is translated to a constraint. All these constraints are fed to the constraint solver. If a model is found, it represents the set of executed method calls and their execution order. This information is extracted from the model and added to the execution graph. Execution order specifies the order of method calls that were not ordered by the program order. The resulting graph is topologically sorted and the resulting trace is shown to the user in a graphical user interface.

E.5 TM Algorithms in Samand

Note that we have restated the algorithms for the number of threads and locations that are needed for the testing program. Also the foreach loops and procedure callas are inlined. The set and map objects are implemented by registers.

The specification is DSTM is as follows:

```
Loc {
   writer: AtomicRegister
   oldValue: AtomicRegister
   newValue: AtomicRegister

   // These could be basic registers.
   // The bug exists even with these stronger registers.
}

DSTM {
   state: AtomicCASRegister[4]   // To store thread
       identifiers
   // Let state[3] be the state of the init trans
   start: AtomicCASRegister[2]   // To store Reference to Loc

   rset: TLocal AtomicRegister[2]
   // This could be a basic register array.
   // The bug exists even with these stronger registers.

   def this() {
      // init state and start
      I01>    state[1].write(\R)
      I02>    state[2].write(\R)
      I03>    state[3].write(\C)
      I04>    loc1 = new Loc()
      I05>    loc1.writer.write(3)
      I06>    loc1.newValue.write(0)
      I07>    start[0].write(loc1)
      I08>    loc2 = new Loc()
      I09>    loc2.writer.write(3)
      I10>    loc2.newValue.write(0)
      I11>    start[1].write(loc2)
   }
```

```
def read(i) {
   R0>     s = state[t].read()
   R1>     if (s = \A)
   R2>        return \A
   R3>     start = start[i].read()
   // --------------
   // Stable value
   R4>     tp = start.writer.read()
   R5>     sp = state[tp].read()
   R6>     if (tp != t && sp = \R)
   R7>        state[tp].cas(\R, \A)
   R8>     if (sp = \A)
   R9>        v = start.oldValue.read()
           else
   R10>       v = start.newValue.read()
   // --------------
   R11>    if (tp != t)
   R12>       rset[i].write(v)
   // --------------
   // Validate
   R13>    rv0 = rset[0].read()
   R14>    if (rv0 != \bot) {
   R15>       s0 = start[0].read()
   R16>       wt0 = s0.writer.read()
   R17>       st0 = state[wt0].read()
   R18>       if (st0 = \C)
   R19>          vp0 = s0.newValue.read()
              else
   R20>          vp0 = s0.oldValue.read()
   R21>       if (rv0 != vp0)
   R22>          return \A
   R23>       cts0 = state[t].read()
   R24>       if (cts0 != \R)
   R25>          return \A
           }
   R26>    rv1 = rset[1].read()
   R27>    if (rv1 != \bot) {
   R28>       s1 = start[1].read()
   R29>       wt1 = s1.writer.read()
   R30>       st1 = state[wt1].read()
   R31>       if (st1 = \C)
   R32>          vp1 = s1.newValue.read()
              else
   R33>          vp1 = s1.oldValue.read()
   R34>       if (rv1 != vp1)
   R35>          return \A
   R36>       cts1 = state[t].read()
   R37>       if (cts1 != \R)
   R38>          return \A
```

```
            }
    // ---------------
    R39>    return v
}

def write(i, v) {
    W0>     s = state[t].read()
    W1>     if (s = \A)
    W2>        return \A
    W3>     start = start[i].read()
    W4>     wt = start.writer.read()
    W5>     if (wt = t) {
    W6>        start.newValue.write(v)
    W7>        return \Ok
            }
    // ---------------
    // Stable value
    W8>     tp = start.writer.read()
    W9>     sp = state[tp].read()
    W10>    if (tp != t && sp = \R)
    W11>       state[tp].cas(\R, \A)
    W12>    if (sp = \A)
    W13>       vp = start.oldValue.read()
            else
    W14>       vp = start.newValue.read()
    // ---------------
    W15>    startp = new Loc()
    W16>    startp.writer.write(t)
    W17>    startp.oldValue.write(vp)
    W18>    startp.newValue.write(v)
    W19>    b = start[i].cas(start, startp)
    W20>    if (b = 1)
    W21>       return \Ok
            else
    W22>       return \A
}

def commit() {
    C01>    rv0 = rset[0].read()
    C02>    if (rv0 != \bot) {
    C03>       s0 = start[0].read()
    C04>       wt0 = s0.writer.read()
    C05>       st0 = state[wt0].read()
    C06>       if (st0 = \C)
    C07>          vp0 = s0.newValue.read()
               else
    C08>          vp0 = s0.oldValue.read()
    C09>       if (rv0 != vp0)
    C10>          return \A
    C11>       cts0 = state[t].read()
```

```
   C12>       if (cts0 != \R)
   C13>          return \A
              }
   C14>       rv1 = rset[1].read()
   C15>       if (rv1 != \bot) {
   C16>          s1 = start[1].read()
   C17>          wt1 = s1.writer.read()
   C18>          st1 = state[wt1].read()
   C19>          if (st1 = \C)
   C20>             vp1 = s1.newValue.read()
                 else
   C21>             vp1 = s1.oldValue.read()
   C22>          if (rv1 != vp1)
   C23>             return \A
   C24>          cts1 = state[t].read()
   C25>          if (cts1 != \R)
   C26>             return \A
              }
   C27>       b = state[t].cas(\R, \C)
   C28>       if (b = 1)
   C29>          return \C
              else
   C30>          return \A
}

main {
   {
      S11>     rset[0].write(\bot)
      S12>     rset[1].write(\bot)

      L11>     v10 = read(0)
      L12>     v11 = read(1)
      L13>     write(0, 7)
      L14>     c1 = commit()
   } || {
      S21>     rset[0].write(\bot)
      S22>     rset[1].write(\bot)

      L21>     v20 = read(0)
      L22>     v21 = read(1)
      L23>     write(1, 7)
      L24>     c2 = commit()
   }
}

order {
   R3 -> R15 &&
   R3 -> R28 &&
   W16 -> W19 &&
   W17 -> W19 &&
```

```
      W18 -> W19 &&
      C02 -> C27 &&
      C15 -> C27
   }

   spec {
      ~(
         L11_Ret \prec L22_Inv /\
         L21_Ret \prec L12_Inv /\

         L12_Ret \prec L23_Inv /\
         L22_Ret \prec L13_Inv /\

         L13_Ret \prec L24_Inv /\
         L23_Ret \prec L14_Inv /\

         v10 = 0 /\
         v11 = 0 /\
         v20 = 0 /\
         v21 = 0 /\
         c1 = \C /\
         c2 = \C
      )
   }
}
```

The specification of McRT is as follows:

```
McRT {

   r: AtomicRegister[2]
   ver: AtomicRegister[2]
   lock: TryLock[2]

   rset: TLocal AtomicRegister[2]
   uset: TLocal AtomicRegister[2]
   // These regs could be basic register arrays

   def this() {
      L01>    lock[0].unlock()
      L02>    lock[1].unlock()
      L03>    r[0].write(0)
      L04>    r[1].write(0)
      L05>    ver[0].write(0)
      L06>    ver[1].write(0)
   }

   def read(i) {
      R0>    u = uset[i].read()
      R1>    if (u = \bot) {
      R2>       ve = ver[i].read()
```

```
   R3>        l = lock[i].read()
   R4>        if (l = 1) {
   R5>           ov0 = uset[0].read()
   R6>           if (ov0 != \bot) {
   R7>              r[0].write(ov0)
   R8>              lock[1].unlock()
                 }
   R9>           ov1 = uset[1].read()
   R10>          if (ov1 != \bot) {
   R11>             r[1].write(ov1)
   R12>             lock[1].unlock()
                 }
   R13>        return \A
            }
   R14>     r = rset[i].read()
   R15>     if (r = \bot)
   R16>        rset[i].write(ve)
         }
   R17>  v = r[i].read()
   R18>  return v
}

def write(i, v) {
   W0>    u = uset[i].read()
   W1>    if (u = \bot) {
   W2>       l = lock[i].tryLock()
   W3>       if (l = 0) {
   W4>          ov0 = uset[0].read()
   W5>          if (ov0 != \bot) {
   W6>             r[0].write(ov0)
   W7>             lock[0].unlock()
                }
   W8>          ov1 = uset[1].read()
   W9>          if (ov1 != \bot) {
   W10>            r[1].write(ov1)
   W11>            lock[1].unlock()
                }
   W12>        return \A
            }
   W13>     ov = r[i].read()
   W14>     uset[i].write(ov)
         }
   W15>  r[i].write(v)
   W16>  return \Ok
}

def commit() {
   C0>    ove0 = rset[0].read()
   C1>    if (ove0 != \bot) {
   C2>       l0 = lock[0].read()
```

```
C3>        ve0 = ver[0].read()
C4>        if ((l0 = 1)|| (ve0 != ove0)) {
C5>           ov10 = uset[0].read()
C6>           if (ov10 != \bot) {
C7>              r[0].write(ov10)
C8>              lock[0].unlock()
              }
C9>           ov11 = uset[1].read()
C10>          if (ov11 != \bot) {
C11>             r[1].write(ov11)
C12>             lock[1].unlock()
              }
C13>          return \A
           }
        }
C14>    ove1 = rset[1].read()
C15>    if (ove1 != \bot) {
C16>       l1 = lock[1].read()
C17>       ve1 = ver[1].read()
C18>       if ((l1 = 1) || (ve1 != ove1)) {
C19>          ov20 = uset[0].read()
C20>          if (ov20 != \bot) {
C21>             r[0].write(ov20)
C22>             lock[0].unlock()
              }
C23>          ov21 = uset[1].read()
C24>          if (ov21 != \bot) {
C25>             r[1].write(ov21)
C26>             lock[1].unlock()
              }
C27>          return \A
           }
        }

C28>    u0 = uset[0].read()
C29>    if (u0 != \bot) {
C30>       v0 = ver[0].read()
C31>       vp0 = v0 + 1
C32>       ver[0].write(vp0)
C33>       lock[0].unlock()
        }
C34>    u1 = uset[1].read()
C35>    if (u1 != \bot) {
C36>       v1 = ver[1].read()
C37>       vp1 = v1 + 1
C38>       ver[1].write(vp1)
C39>       lock[1].unlock()
        }

C40>    return \C
```

```
   }

   main {
      {
         I11>    rset[0].write(\bot)
         I12>    rset[1].write(\bot)
         I13>    uset[0].write(\bot)
         I14>    uset[1].write(\bot)

         L11>   r1 = read(1)
         L12>   write(0, 7)
         L13>   c1 = commit()
      } || {
         I21>    rset[0].write(\bot)
         I22>    rset[1].write(\bot)
         I23>    uset[0].write(\bot)
         I24>    uset[1].write(\bot)

         L21>   write(1, 7)
         L22>   r2 = read(0)
         L23>   c2 = commit()
      }
   }

   order {
      R2 -> R3 &&
      R3 -> R17 &&
      C2 -> C3 &&
      C16 -> C17
   }

   spec {
      ~(
         r1 = 7  /\
         r2 = 7  /\
         c1 = \A /\
         c2 = \A
      )
   }
}
```

F On the Definition of Local Progress

We have sharpened the definition of crashing from
A process T is *crashing* in an infinite history H if $H|T$ is a finite sequence of operations.
to
A process T is *crashing* in an infinite history H if $H|T$ is a finite sequence of operations (not ending in an abort $ret_T(\mathbb{A})$ or commit $ret_T(\mathbb{C})$ response event).

Theorem 2 (from the original paper) continues to be correct and its proof can be completed as follows:
The proof for the branch "Sys is crash-prone" considers two cases: (1) Process p_1 crashes in history H. (2) Process p_1 does not crash in history H. There will be an additional subcase for case 2 above.
Subcase: p_1 ends in $x.read^1() \cdot A^1$.
The proof of this subcase is similar to the proof of case 1. Process $H|p_1$ is finite thus, process p_2 is pending and invokes infinitely many operations. Process p_2 invokes infinitely many operations iff the strategy executes infinitely many iterations of Step 2. At each iteration of Step 2 process p_2 either receives abort event or invokes commit, thus p_2 is correct in H. Since M ensures local progress and p_2 is infinite and correct in H, then process p_2 is not pending: a contradiction.