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:

$$\begin{aligned} P_{WS} &= \{read_{T_1}(1){:}r_{11}\ \ read_{T_1}(2){:}r_{12}\ \ write_{T_1}(0,v_1)\ \ commit_{T_1}(){:}c_1\}\ ||\\ &\quad\ \{read_{T_2}(1){:}r_{21}\ \ read_{T_2}(2){:}r_{22}\ \ write_{T_2}(1,v_1)\ \ commit_{T_2}(){:}c_2\}\\ WS &= (r_{11}=v_0\ \wedge\ r_{12}=v_0\ \wedge\ r_{21}=v_0\ \wedge\ r_{22}=v_0\ \wedge\ c_1=\mathbb{C}\ \wedge\ c_2=\mathbb{C}) \end{aligned}$$

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** $(t \neq t')$ $state[t'].cas(\mathbb{R}, \mathbb{A})$ }

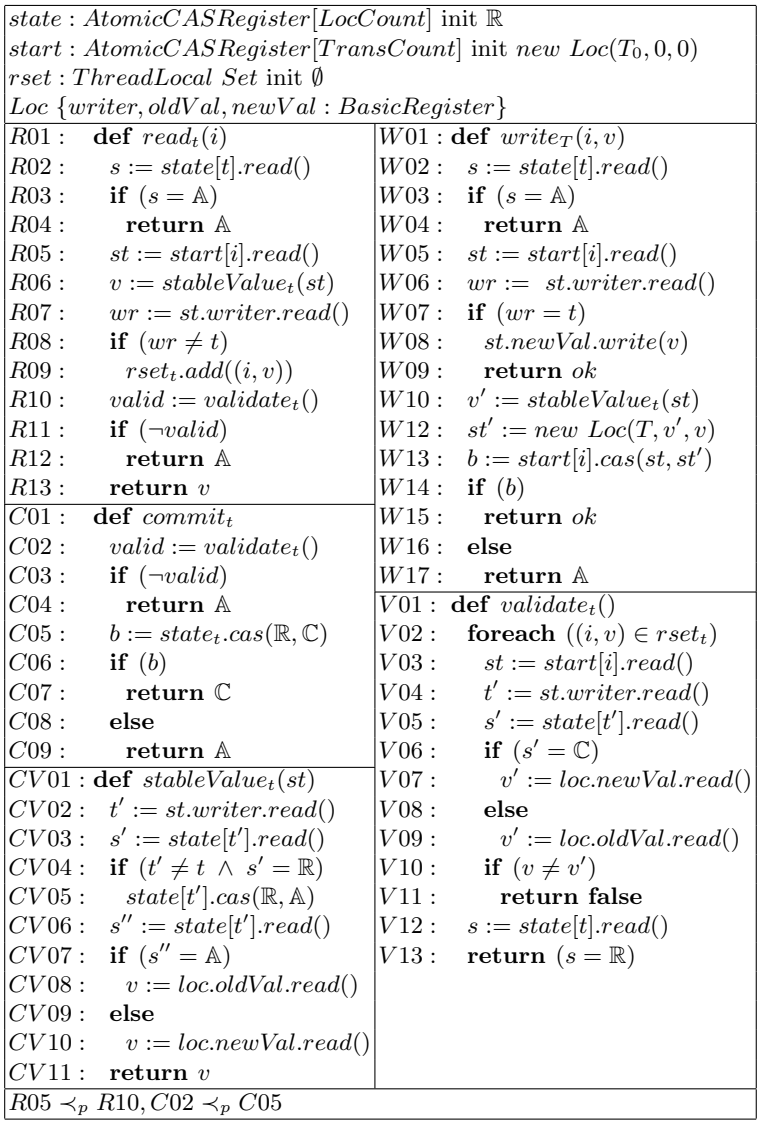

Fig. 1. Core DSTM

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

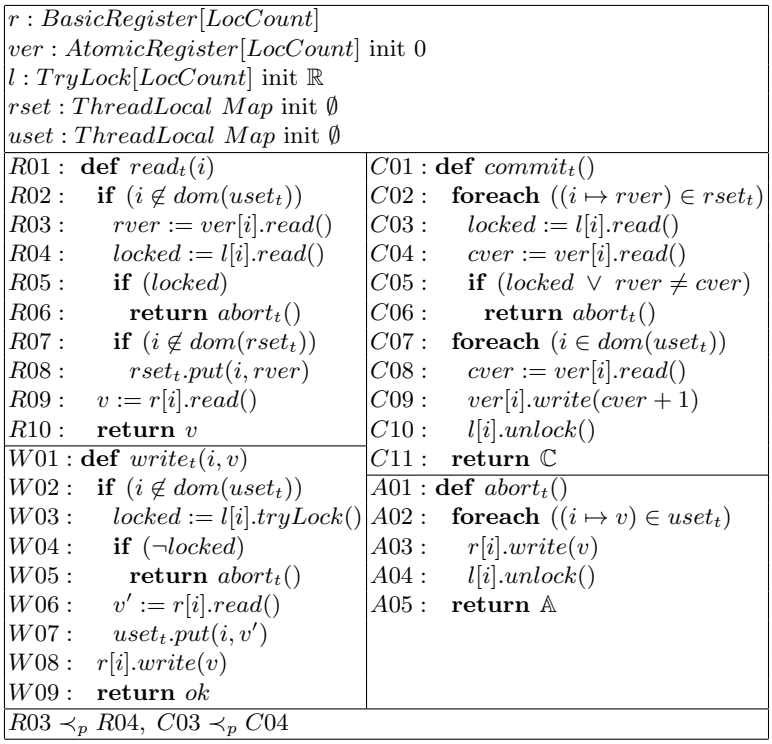

Fig. 2. Core McRT

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.