Putting Opacity in its Place

Mohsen Lesani
Computer Science Department
University of California, Los Angeles, CA

Victor Luchangco Mark Moir
Oracle Labs
Burlington, MA, USA

ABSTRACT

We clarify the relationships between Guerraoui and Kapalka's opacity correctness condition for Transactional Memory (TM) algorithms and the TMS1 and TMS2 conditions we have previously proposed. Using formal, machine checked simulation proofs constructed using the PVS theorem proving system, we have shown that all algorithms that satisfy opacity also satisfy TMS1, and that all algorithms that satisfy TMS2 also satisfy opacity.

1. INTRODUCTION

Transactional memory (TM) [8, 16] provides an abstraction that allows programmers to express that a block of code should appear to be executed atomically (either all or none of its effects should be seen) and in isolation (transactions should not observe each other's partial effects). Guaranteeing these properties is the responsibility of the system (some combination of compiler, runtime library, and hardware), not the programmer. Thus, TM aims to bring similar benefits to shared memory programmers as database transactions have delivered to database programmers for decades.

A dizzying array of TM algorithms have been proposed in the quest to achieve good performance and scalability. Many of these algorithms are complex and subtle, particularly because they execute transactions "optimistically": they execute transactions concurrently in the hope that no conflicts occur, but are prepared to detect conflicts that do occur. If no conflicts occur, a transaction's effects can become visible to others, in which case it is said to *commit*; otherwise, conflicts may be resolved by ensuring that a transaction's effects do not become visible, in which case it is said to *abort*.

To be useful, it is crucial that TM algorithms correctly provide the guarantees expected by programmers. Despite all of the work on many different TM algorithms, relatively little attention has been paid to formally specifying what it means for them to be correct, a critical step on the path to proving that they are correct.

Different requirements are appropriate for different contexts. In our work, we have been most interested in specifying correctness properties for TM runtime libraries to be used to implement proposed transactional language features for C++ [1]. In this context, because user code is executed during transactions, it is important that no transaction observes inconsistent behavior *even if it ultimately aborts*, because this can cause fatal runtime errors such as divide-by-zero, as well as infinite loops. For this reason, traditional correctness conditions for database transactions—such as *serializability* [13]—are not adequate for this context because they say nothing about the behavior of transactions that abort.

Guerraoui and Kapalka [6] were the first to attempt to formalize a correctness condition—*opacity*—that requires all transactions (including active and aborted ones) to observe consistent behavior. Opacity requires that, at any time, there is a *single* execution that is consistent with the behavior observed by *all* transactions. We and others [4, 9] have previously observed that this requirement is unnecessarily restrictive: provided a transaction observes behavior that is consistent with *some* correct execution, fatal errors while executing user code are avoided. In fact, opacity precludes some eligible implementations such as dependence-aware [2, 15] TM algorithms.

In our previous work [4], we defined a condition TMS1 that requires the behavior observed by all committed transactions to be justified by a single execution, while allowing active and aborted ones to be justified by different executions. We specified TMS1 precisely as the set of executions exhibited by an I/O automaton, which we defined and modeled using the PVS langauge [14], allowing for formal, machine-checked proofs that TM algorithms satisfy TMS1 via well established proof methods such as simulation and refinement [11, 12].

To our knowledge, no rigorous proof that a TM algorithm satisfies opacity has been developed prior to our work in this area. In particular, the most commonly used method used to show that a TM algorithm satisfies opacity depends on structural properties of the algorithm [5], which have not been formally shown to hold for any TM algorithm as far as we know.

We have another paper under submission [10], in which we present a framework we have developed using the PVS theorem prover [14] for proving that TM algorithms (mod-

eled using I/O automata) satisfy various correctness conditions (also modeled using I/O automata). We have used this framework to construct a formal, machine-checked hierarchical proof that the NOrec TM algorithm [3] satisfies TMS1. One of the automata in this proof hierarchy models the TMS2 condition presented in [4]. TMS2 is more restrictive than TMS1, but is much closer to the intuition of many TM algorithms, including NOrec. Thus, it is easier to prove that NOrec satisfies TMS1 by proving that it satisfies TMS2, and composing this result with our proof that TMS2 satisfies TMS1.

We have recently reproved the latter result in such a way that it has a side effect of "putting opacity in its place". Specifically, we proved that TMS2 satisfies TMS1 in two steps by proving that TMS2 satisfies opacity and that opacity satisfies TMS1. This result confirms our previous conjecture [4], which is good news for several reasons. First, algorithms that are proved to satisfy opacity are now known to also satisfy TMS1. Second, our result paves the way for rigorous machine-checked proofs that TM algorithms satisfy opacity, and in fact, to our knowledge, our proofs that NOrec satisfies TMS2 and that TMS2 satisfies opacity together constitute the first such proof. We also find it comforting to confirm our belief that accepting TMS1 as the correctness condition of choice for a class of TM algorithms does not require us to preclude any algorithms that satisfy opacity, which has been widely accepted by the community.

In the remainder of the paper, we introduce the I/O automaton that formally specifies the opacity condition, and give an overview of the proofs establishing that TMS2 satisfies opacity and that opacity satisfies TMS1.

2. OPACITY AS AN IOA

Because an I/O automaton cannot generate an execution without first generating all of its prefixes, the set of executions of every I/O automaton is naturally prefix-closed. For the same reason, an implementation that satisfies opacity cannot exhibit an execution that has a prefix that does not satisfy opacity. Some definitions of opacity [6] have nonetheless allowed such executions. By specifying opacity via an I/O automaton whose executions are naturally prefix closed, we model the prefix-closed definition of opacity [7].

Figure 1 presents the automaton we use to precisely specify $\mathsf{Opacity}(\mathcal{O})$, the opacity condition for a TM algorithm implementing transactions on a generic object $\mathcal{O}$, whose sequential semantics are represented by $legal_{\mathcal{O}}$, the set of legal sequences of operations on $\mathcal{O}$. In this figure, $\sigma|_S$ denotes the subsequence of the sequence σ containing only elements of the set S. $\sigma|_{\leq s}$ denotes the prefix of the sequence σ up to and including the element s.

The actions in the lefthand column of Figure 1 represent invocations by transactions, with the preconditions capturing well-formedness requirements, and the effects making appropriate state transitions that should be self explanatory. Two points that may require further explanation concern the effects of the begin_t and $\mathsf{inv}_t(i)$ actions. The begin_t action adds a pair (t', t) to *extOrder* for every transaction t' that completed before t began; this records the real-time ordering between transactions, so that we can enforce opacity's requirement that the order in which transactions appear to take effect preserves the real-time order of transactions. The $\mathsf{inv}_t(i)$ action records t's invocation, so that—together with the corresponding response—it can be validated and recorded when the response occurs.

The actions in the righthand column represent the TM system's responses to transactions' invocations. Again, the preconditions capture well-formedness requirements that should be self-explanatory, and the effects make appropriate state transitions; note that $\mathsf{resp}_t(r)$ records t's pending operation together with the response r in t's operations.

It remains to describe the most interesting and important parts of the automaton, which are the key to ensuring that it produces exactly the set of opaque histories.

The preconditions of the $\mathsf{resp}_t(r)$, $\mathsf{commitOk}_t$, and abort_t actions each have an additional requirement whose purpose is to ensure that, after the action takes effect, the history produced so far by the automaton is opaque. (We also specified an automaton that expresses opacity in the *postconditions* of the actions, which is slightly simpler and more intuitively maps to opacity, but is less convenient for proving the properties stated in the next section. We have proved the two formulations to be equivalent.) Let us first examine the precondition for $\mathsf{commitOk}_t$.

The *opaqueValidCommit*(t) condition requires that there exists a serialization σ of all transactions that have started, such that σ respects the real-time order of transactions (denoted by $ser(ST, extOrder)$). It further requires that there is a set S that contains all transactions that have committed, some subset of those that are commit-pending (i.e., have invoked commit, but have neither committed nor failed yet), and t itself, as t will have committed after the action takes effect. Finally, it requires that, for every transaction $t' \in S$, the history that is produced by concatenating the operations performed by each of the transactions in S up to and including t' in the order of σ respects the sequential semantics of the implemented object $\mathcal{O}$. We note that, although all transactions before t' are considered to be committed (either they are actually committed, or they are commit-pending and have been chosen for the set S), t' itself may not be committed and may not even have invoked commit. This captures the requirement that the serialization chosen also justifies the responses of active transactions.

This captures the same requirements as stated for opacity in [6]:

> A history H is opaque if there exists a sequential history S equivalent to some history in set $Complete(H)$, such that (1) S preserves the real-time order of H, and (2) every transaction $T_i \in S$ is legal in S.

The precondition for the abort_t action is identical, except that it requires that t is *not* included in the set of transactions chosen, because t will abort if the action takes effect. The precondition for $\mathsf{resp}_t(r)$ is again similar, but it modifies the value of ops_t to include the invocation-response pair that

State variables

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

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

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

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

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

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

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

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

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

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

Derived state variables, functions and predicates

$$ST \triangleq \{t \mid status_t \neq \mathsf{notStarted}\}$$
$$DT \triangleq \{t \mid status_t \in \{\mathsf{committed}, \mathsf{aborted}\}\}$$
$$CT \triangleq \{t \mid status_t = \mathsf{committed}\}$$
$$CPT \triangleq \{t \mid status_t = \mathsf{commitPending}\}$$
$$opSeq(\sigma) \triangleq ops_{t_1} \cdot ops_{t_2} \cdot \ldots \cdot ops_{t_n} \text{ where } \sigma = t_1 t_2 \ldots t_n$$

$$opaque(\sigma, S, ops) \triangleq \forall t \in range(\sigma): legal_\mathcal{O}(opSeq(\sigma|_{S \cup \{t\}}|_{\leq_t}, ops))$$
$$opaqueValidCommit(t) \triangleq \exists \sigma, S: \sigma \in ser(ST, extOrder) \wedge CT \cup \{t\} \subseteq S \subseteq CT \cup CPT \wedge opaque(\sigma, S, ops)$$
$$opaqueValidFail(t) \triangleq \exists \sigma, S: \sigma \in ser(ST, extOrder) \wedge CT \subseteq S \subseteq CT \cup CPT \setminus \{t\} \wedge opaque(\sigma, S, ops)$$
$$opaqueValidResp(t, i, r) \triangleq \exists \sigma, S: \sigma \in ser(ST, extOrder) \wedge CT \subseteq S \subseteq CT \cup CPT \wedge opaque(\sigma, S, ops[ops_t := ops_t \cdot (i, r)])$$

Figure 1: Opacity($\mathcal{O}$) Automaton.

will become part of the history if the action takes effect, so must be included in the evaluation of whether the history will be opaque after the action takes effect.

3. OVERVIEW OF PROOFS

In this short paper, we describe the proofs that "put opacity in its place" only at a high level. We plan to release our proof framework, which includes these proofs, in the near future, hopefully in conjunction with the above-mentioned paper about the framework [10]. At that point, interested readers will be able to view the definitions and properties used in our proof in precise detail, and will even be able to step interactively through our proofs using PVS.

In [4], we specified our TMS1 condition via an automaton TMS1($\mathcal{O}$) that captures the allowed behavior of a TM algorithm implementing transactions on a generic object $\mathcal{O}$. However, because TMS2 is intended to be closer to the intuition of common TM algorithms that typically implement a shared memory object, we specified TMS2 directly for such objects, via a TMS2(*init*) automaton (where *init* is a predicate that is assumed to be satisfied by the initial state of the shared memory object).

Using our framework, we have formally specified and mechanically proved the following two theorems, where $A \leq_F B$ denotes that there is a forward simulation from automaton A to automaton B.

Theorem 1 $\mathsf{Opacity}(\mathcal{O}) \leq_F \mathsf{TMS1}(\mathcal{O})$.

Theorem 2 $\mathsf{TMS2}(\mathit{init}) \leq_F \mathsf{Opacity}(\mathit{Mem}(\mathit{init}))$.

The automaton $\mathsf{Opacity}(\mathit{Mem}(\mathit{init}))$ is $\mathsf{Opacity}(\mathcal{O})$ instantiated with a read-write memory object whose initial state satisfies the *init* predicate.

The most interesting and challenging part of proving these forward simulations is showing that, when the precondition of the $\mathsf{resp}_t(r)$, $\mathsf{commitOk}_t$, or abort_t action holds in a state of the more concrete automaton (e.g., $\mathit{opacity}(\mathcal{O})$ in the case of Theorem 1), then it also holds in any state of the more abstract automaton ($\mathsf{TMS1}(\mathcal{O})$ for Theorem 1) that is related by the simulation relation.

$\mathsf{TMS2}(\mathit{init})$ is expressed using a sequence of memory states [4], and preconditions for the above-mentioned actions are stated in terms of validity with respect to that sequence. For example, reads performed by writing transactions that commit successfully must be consistent with the last memory state in the sequence at the time they take effect, while reads of read-only transactions are allowed to be consistent with any memory state in the sequence that was the last one at any point during the read-only transaction's execution. As a result, there is somewhat of an intuitive gap between the preconditions in the commit actions of $\mathsf{TMS2}(\mathit{init})$ and $\mathsf{Opacity}(\mathit{Mem}(\mathit{init}))$.

We addressed this issue by performing the proof for Theorem 2 in two steps. We defined another automaton $\mathsf{TxnOrdTMS2}(\mathit{init})$, which records the initial state of the memory and a sequence of committed writer transactions (in the order that they commit) instead of the sequence of memory states that those transactions write. We then proved the following two lemmas, which together imply Theorem 2.

Lemma 1 $\mathsf{TMS2}(\mathit{init}) \leq_F \mathsf{TxnOrdTMS2}(\mathit{init})$.

Lemma 2 $\mathsf{TxnOrdTMS2}(\mathit{init}) \leq_F \mathsf{Opacity}(\mathit{Mem}(\mathit{init}))$.

4. REFERENCES

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

[2] Utku Aydonat and Tarek Abdelrahman. Serializability of transactions in software transactional memory. In *TRANSACT '08: 3rd Workshop on Transactional Computing*, feb 2008.

[3] Luke Dalessandro, Michael F. Spear, and Michael L. Scott. NOrec: Streamlining STM by abolishing ownership records. In *PPoPP'10: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, January 2010.

[4] Simon Doherty, Lindsay Groves, Victor Luchangco, and Mark Moir. Towards formally specifying and verifying transactional memory. *Formal Aspects of Computing*, 2012.

[5] Rachid Guerraoui, Thomas A. Henzinger, Barbara Jobstmann, and Vasu Singh. Model checking transactional memories. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 372–382, 2008.

[6] Rachid Guerraoui and Michal Kapalka. On the correctness of transactional memory. In *PPoPP'08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 175–184, 2008.

[7] Rachid Guerraoui and Michal Kapalka. *Principles of Transactional Memory*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2010.

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

[9] Damien Imbs, José Ramon de Mendivil, and Michel Raynal. Brief announcement: virtual world consistency: a new condition for stm systems. In *Proceedings of the 28th ACM symposium on Principles of distributed computing*, PODC '09, pages 280–281, New York, NY, USA, 2009. ACM.

[10] Mohsen Lesani, Victor Luchangco, and Mark Moir. A framework for formally specifying and verifying the correctness of transactional memory algorithms. under submission, March 2012.

[11] N. Lynch and M. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, pages 137–151, August 1987.

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

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

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

[15] Hany E. Ramadan, Christopher J. Rossbach, and Emmett Witchel. Dependence-aware transactional memory for increased concurrency. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 41, pages 246–257, Washington, DC, USA, 2008. IEEE Computer Society.

[16] Nir Shavit and Dan Touitou. Software transactional memory. In *ACM Symposium on Principles of Distributed Computing*, pages 204–213. ACM Press, 1995.