Safety of Live Transactions in Transactional Memory: TMS is Necessary and Sufficient[*]

Hagit Attiya[*] Alexey Gotsman[†] Sandeep Hans[*] Noam Rinetzky[‡]

May 13, 2014

Abstract

The main challenge of stating the correctness of transactional memory (TM) systems is the need to provide guarantees on the system state observed by *live* transactions, i.e., those that have not yet committed or aborted. A TM correctness condition should not be too restrictive, to allow flexibility in implementation, yet strong enough to disallow undesirable TM behavior, which can lead to run-time errors in live transactions. The latter feature is formalized by *observational refinement* between TM implementations, stating that properties of a program using a concrete TM implementation can be established by analyzing its behavior with an abstract TM, serving as a specification of the concrete one.

We show that a variant of *transactional memory specification* (*TMS*) is equivalent to observational refinement for the common programming model in which local variables are rolled back upon a transaction abort and, hence, is the weakest consistency condition for this case. This is challenging due to the nontrivial formulation of TMS, which allows different aborted and live transactions to have different views on the system state. Our proof reveals some natural, but subtle, assumptions on the TM required for the equivalence result.

1 Introduction

Transactional memory (TM) eases the task of writing concurrent applications by letting the programmer designate certain code blocks as *atomic*. TM allows developing a program and reasoning about its correctness as if each atomic block executes as a *transaction*—in one step and without interleaving with others—even though in reality the blocks can be executed concurrently. Figure 1 shows how atomic blocks yield simple code for computations involving several shared *transactional objects* X, Y and Z, access to which is mediated by the TM.

```
result := abort;
while (result == abort) {
   result := atomic {
      x = X.read();
      y = Y.read();
      z = 42 / (x - y);
      Z.write(z);
   } }
```

Figure 1: TM usage

The common approach to stating TM correctness is through a *consistency condition* that restricts the possible TM executions. The main subtlety of formulating such a condition is the need to provide guarantees on the state of transactional objects observed by *live* transactions, i.e., those that have not yet committed or aborted. Because live transactions can always be aborted, one might think it unnecessary to provide any guarantees for them, as in fact done by common database consistency conditions [9]. However, in the setting of transactional memory, this is often unsatisfactory. For example, in Figure 1 the programmer may rely on the fact that $X \neq Y$, and, correspondingly,

[*]Technion - Israel Institute of Technology, Israel, {hagit,sandeep}@cs.technion.ac.il
[†]IMDEA Software Institute, Spain, alexey.gotsman@imdea.org
[‡]Tel Aviv University, Israel, maon@cs.tau.ac.il
[*]A longer version of this paper will be submitted to DISC; the full version will appear on the authors' webpage.

make sure that every committing transaction preserves this invariant. If we allow the transaction to read values of X and Y violating the invariant (counting on it to abort later, due to inconsistency), this will lead to the program *faulting* due to a division by zero.

The question of which TM consistency condition to use is far from settled, with several candidates having been proposed [2–4, 7]. An ideal condition should be as weak as possible, to allow flexibility in TM implementations, yet strong enough to satisfy the intuitive expectations of the programmer and, in particular, to disallow undesirable behaviors such as the one described above. *Observational refinement* [5, 6] allows formalizing the programmer's expectations and thereby evaluating consistency conditions systematically. Consider two TM implementations—a *concrete* one, such as an efficient TM, and an *abstract* one, such as a TM executing every atomic block atomically. Informally, the concrete TM observationally refines the abstract one for a given programming language if every behavior a user can observe of a program P in this language linked with the concrete TM can also be observed when P is linked with the abstract TM instead. This allows the programmer to reason about the behavior of P (e.g., the preservation of the invariant $\mathsf{X} \neq \mathsf{Y}$) using the expected intuitive semantics formalized by the abstract TM; the observational refinement relation implies that the conclusions (e.g., the safety of the division in Figure 1) will carry over to the case when P uses the concrete TM.

In prior work [1] we showed that a variant of the *opacity* condition [4] coincides with observational refinement for a particular programming language and, hence, is the weakest consistency condition for this language. Roughly speaking, a concrete TM implementation is in the opacity relation with an abstract one if for any sequence of interactions with the concrete TM, dubbed a *history*, there exists a history of the abstract TM where: (i) the actions of every separate thread are the same as in the original history; and (ii) the order of non-overlapping transactions present in the original history is preserved. However, our result considered a programming language in which local variables modified by a transaction are not rolled back upon an abort. Although this assumption holds in some situations (e.g., for Scala STM [10]), it is non-standard and most TM systems do not satisfy it.

2 Our Contributions

In this paper, we consider a variant of *transactional memory specification* (*TMS*) [3], a condition weaker than opacity,[*] and show that, under some natural assumptions on the TM, it coincides with observational refinement for a programming language in which local variables do get rolled back upon an abort.

This result is not just a straightforward adjustment of the one about opacity to a more realistic setting: TMS weakens opacity in a nontrivial way, which makes reasoning about its relationship with observational refinement much more intricate. In more detail, the key feature of opacity is that the behavior of *all* transactions in a history of the concrete TM, including aborted and live ones, has to be explained by a single history of the abstract TM. TMS relaxes this requirement by requiring only committed transactions in the concrete history to be explained by a single abstract one obeying (i)–(ii) above; every response obtained from the TM in an aborted or live transaction may be explained by a separate abstract history. The constraints on the choice of the abstract history are subtle: on one hand, somewhat counterintuitively, TMS allows it to include transactions that aborted in the concrete history, with their status changed to committed, and exclude some that committed; on the other hand, this is subject to certain carefully chosen constraints. The flexibility in the choice of the abstract history is meant to allow the concrete TM implementation to perform as many optimizations as possible. However, it is not straightforward to establish that this flexibility does not invalidate

[*]The condition we present here is actually called TMS1 in [3, 8]. These papers also propose another condition, called TMS2, but it is stronger than opacity [8] and therefore not considered here.

observational refinement (and hence, the informal guarantees that programmers expect from a TM) or that the TMS definition cannot be weakened further.

Our results ensure that this is indeed the case. Informally, if local variables are not rolled back when transactions abort, threads can communicate to each other the observations they make inside aborted transactions about the state of transactional objects. This requires the TM to provide a consistent view of this state across all transactions, as formalized by the use of a single abstract history in opacity. However, if local variables are rolled back upon an abort, no information can leak out of an uncommitted transaction, possibly apart from the fact that the code in the transaction has faulted, stopping the computation. To get observational refinement in this case we only need to make sure that a fault in the transaction occurring with the concrete TM could be reproduced with the abstract one. For this it is sufficient to require that the state of transactional objects seen by every live transaction can be explained by some abstract history; different transactions can be explained by different histories.

Technically, we prove that the TMS relation between a concrete TM and an abstract TM is sufficient for observational refinement. This is proved by establishing a nontrivial property of the set of computations of a program, showing that a live transaction cannot notice the changes in the committed/aborted status of transactions concurrent with it that are allowed by TMS. Proving that the TMS relation is necessary for observational refinement is challenging as well, as this requires us to devise multiple programs that can observe whether the subtle constraints governing the change of transaction status in TMS are fulfilled by the TM. We have identified several closure properties on the set of histories produced by the abstract TM required for these results to hold. Although intuitive, these properties are not necessarily provided by an arbitrary TM, and our results demonstrate their importance.

The programming language we consider does not allow explicit transaction aborts or transaction nesting and assumes a static separation of transactional and non-transactional shared memory. Extending our development to lift these restrictions is an interesting avenue for future work.

3 Main Result

The main result of this paper is that the TMS relation can be characterized in terms of observational refinement for a class of TMs that enjoy certain natural closure properties, informally stated as follows:

- A TM is ***closed under immediate aborts*** if it allows adding arbitrary ***immediately aborted*** transactions to any history.
- A TM is ***closed under removing transaction responses*** if it allows removing a TM response by a transaction that does not take further steps.
- A TM is ***closed under removing live and aborted transactions*** if it allows to remove aborted and live transactions.
- A TM is ***closed under commit-pending transactions completion*** if it has a way to complete the commit-pending transactions of any history it contains which has at most one live transaction.

These properties are satisfied by the natural TM specification that executes every transaction atomically [1].

THEOREM 1 (Informal statement). *Let $\mathcal{T}_C$ and $\mathcal{T}_A$ be transactional memories.*

(i) If $\mathcal{T}_A$ is closed under immediate aborts and removing transaction responses, then $\mathcal{T}_C$ is in the TMS relation with $\mathcal{T}_A$ implies that $\mathcal{T}_C$ operationally refines $\mathcal{T}_A$.

(ii) If $\mathcal{T}_A$ is closed under removing live and aborted transactions and commit-pending transactions completion, then $\mathcal{T}_C$ operationally refines $\mathcal{T}_A$ implies that $\mathcal{T}_C$ is in the TMS relation with $\mathcal{T}_A$.

4 Related Work

When presenting TMS [3], Doherty et al. discuss why it allows programmers to think only of serial executions of their programs, in which the actions of a transaction appear consecutively. This discussion—corresponding to our sufficiency result—is informal, since the paper lacks a formal model for programs and their semantics. Most of it explains how to ensure the correctness of committed transactions. The discussion of the most challenging case of live transactions is one paragraph long. It only roughly sketches the construction of a trace with an abstract history allowed by TMS and does not give any reasoning for why this trace is a valid one, but only claims that constraints in the TMS definition ensure this. This reasoning is very delicate, as indicated by our proof, which carefully selects which actions to erase when transforming the trace. Moreover, Doherty et al. do not try to argue that TMS is the weakest condition possible, as we established by our necessity result.

Another TM consistency condition, weaker than opacity but incomparable to TMS, is *virtual world consistency* (*VWC*) [7]. Like TMS, VWC allows every operation in a live or aborted transaction to be explained by a separate abstract history. However, it places different constraints on the choice of abstract histories, which do not take into account the real-time order between actions. Because of this, VWC does not imply observational refinement for our programming language: taking into account the real-time order is necessary when threads can communicate via global variables outside transactions.

Our earlier paper [1] has laid the groundwork for relating TM consistency and observational refinement, and it includes a detailed comparison with related work on opacity and observational refinement. The present paper considers a much more challenging case of a language where local variables are rolled back upon an abort. To handle this case, we have developed new techniques, such as establishing the live transaction insensitivity property to prove sufficiency and proposing monitor programs for the nontrivial constraints used in the TMS definition to prove necessity.

References

[1] H. Attiya, A. Gotsman, S. Hans, and N. Rinetzky. A programming language perspective on transactional memory consistency. *PODC* 2013, pp. 309–318.

[2] H. Attiya, S. Hans, P. Kuznetsov, and S. Ravi. Safety of deferred update in transactional memory. *ICDCS* 2013.

[3] S. Doherty, L. Groves, V. Luchangco, and M. Moir. Towards formally specifying and verifying transactional memory. *Formal Aspects of Computing*, 25(5):769–799, 2013.

[4] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. *PPOPP* 2008, pp. 175–184.

[5] J. He, C. Hoare, and J. Sanders. Data refinement refined. *ESOP* 1986, pp. 187–196.

[6] J. He, C. Hoare, and J. Sanders. Prespecification in data refinement. *Information Processing Letters*, 25(2):71 – 76, 1987.

[7] D. Imbs and M. Raynal. Virtual world consistency: A condition for STM systems (with a versatile protocol with invisible read operations). *Theor. Comput. Sci.*, 444:113–127, 2012.

[8] M. Lesani, V. Luchangco, and M. Moir. Putting opacity in its place. *WTTM* 2012.

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

[10] Scala STM Expert Group. Scala STM quick start guide, 2012. http://nbronson.github.io/scala-stm/quick_start.html.