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

Hagit Attiya[1], Alexey Gotsman[2], Sandeep Hans[1], and Noam Rinetzky[3]

[1] Technion - Israel Institute of Technology, Israel
[2] IMDEA Software Institute, Spain
[3] Tel Aviv University, Israel

Abstract. One of the main challenges in 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 be weak enough 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*), a TM correctness condition, 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 acceptable 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 of 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 are used to manipulate 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); } }
```

Fig. 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 done by common database consistency conditions [1]. However, in the setting of transactional memory, this is often unsatisfactory. For example, in Figure 1 the programmer may rely on the fact that $\mathsf{X} \neq \mathsf{Y}$, and, correspondingly, 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 [2–5]. An ideal condition should be weak enough 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* [6, 7] 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 any program P in this language using the concrete TM can also be observed when P uses 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 [8] we showed that a variant of the *opacity* condition [2] is equivalent to observational refinement for a particular programming language and, hence, is the weakest acceptable 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., Scala STM [9]), it is non-standard and most TM systems do not satisfy it. In this paper, we consider a variant of *transactional memory specification* (*TMS*) [5], a condition weaker than opacity,[4] and show that, under some natural assumptions on the TM, it is equivalent to 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 justified by a single history of the abstract TM. TMS relaxes this requirement by requiring only committed transactions in the concrete history to be justified by a single abstract one obeying (i)–(ii) above; every response obtained from the TM in an aborted or live transaction may be justified by a separate abstract history. The constraints on the choice of the abstract history are subtle: on one hand, somewhat counter-intuitively, 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.

[4] The condition we present here is actually called TMS1 in [5, 10]. These papers also propose another condition, TMS2, but it is stronger than opacity [10] and therefore not considered here.

However, it is not straightforward to establish that this flexibility does not invalidate 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 justified by some abstract history; different transactions can be justified by different histories.

Technically, we prove that TMS is sufficient for observational refinement 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 (Lemma 1, Section 6.1). Proving that TMS 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.

To concentrate on the core goal of this paper, 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. Also, due to space constraints, we defer some of the proofs to Appendix D.

2 Programming Language Syntax

We consider a language where a program $P = C_1 \parallel \cdots \parallel C_m$ is a parallel composition of ***threads*** C_t, $t \in \mathsf{ThreadID} = \{1, \ldots, m\}$. Every thread $t \in \mathsf{ThreadID}$ has a set of ***local variables*** $\mathsf{LVar}_t = \{x, y, \ldots\}$ and threads share a set of ***global variables*** $\mathsf{GVar} = \{g, \ldots\}$, all of type integer. We let $\mathsf{Var} = \mathsf{GVar} \uplus \biguplus_{t=1}^{m} \mathsf{LVar}_t$ be the set of all program variables. Threads can also access a transactional memory, which manages a fixed collection of ***transactional objects*** $\mathsf{Obj} = \{o, \ldots\}$, each with a set of ***methods*** that threads can call. For simplicity, we assume that each method takes one integer parameter and returns an integer value, and that all objects have the same set of methods $\mathsf{Method} = \{f, \ldots\}$. The syntax of commands C is standard: C can be of the forms

$$c \mid C; C \mid \texttt{while}\,(b)\,\texttt{do}\,C \mid \texttt{if}\,(b)\,\texttt{then}\,C\,\texttt{else}\,C \mid x := \texttt{atomic}\,\{C\} \mid x := o.f(e)$$

where b and e denote Boolean and integer expressions over local variables, left unspecified. The syntax includes ***primitive commands*** c from a set PComm, sequential

composition, conditionals, loops, `atomic` blocks and object method invocations. Primitive commands execute atomically, and they include assignments to local and global variables and a special `fault` command, which stops the execution of the program in an error state. Thus, `fault` encodes illegal computations, such as division by zero.

An ***atomic block*** $x := \texttt{atomic}\,\{C\}$ executes C as a ***transaction***, which the TM can ***commit*** or ***abort***. The system's decision is returned in the local variable x, which gets assigned distinguished values committed or aborted. We do not allow programs in our language to abort a transaction explicitly and forbid nested atomic blocks and, hence, nested transactions. We also assume that a program can invoke methods on transactional objects only inside atomic blocks and access global variables only outside them. Local variables can be accessed in both cases; however, threads cannot access local variables of other threads. Due to space constraints, we defer the formalisation of the rules on variable accesses to Appendix A. When we later define the semantics of our programming language, we mandate that, if a transaction is aborted, local variables are rolled back to the values they had at its start, and hence, the values written to them by the transaction cannot be observed by the following non-transactional code.

3 Model of Computations

To define the notion of observational refinement for our programming language and the TMS consistency condition, we need a formal model for program computations. To this end, we introduce *traces*, which are certain finite sequences of *actions*, each describing a single computation step (we do not consider infinite computations).

Definition 1. *Let* ActionId *be a set of action identifiers. A* ***TM interface action*** ψ *has one of the following forms:*

Request actions	*Matching response actions*
$(a, t, \mathsf{txbegin})$	$(a, t, \mathsf{OK}) \mid (a, t, \mathsf{aborted})$
$(a, t, \mathsf{txcommit})$	$(a, t, \mathsf{committed}) \mid (a, t, \mathsf{aborted})$
$(a, t, \mathsf{call}\ o.f(n))$	$(a, t, \mathsf{ret}(n')\ o.f) \mid (a, t, \mathsf{aborted})$

where $a \in \mathsf{ActionId}$*,* $t \in \mathsf{ThreadID}$*,* $o \in \mathsf{Obj}$*,* $f \in \mathsf{Method}$ *and* $n, n' \in \mathbb{Z}$*. A* ***primitive action*** χ *has the form* (a, t, c)*, where* $c \in \mathsf{PComm}$ *is a primitive command. We use* φ *to range over actions of either type.*

TM interface actions denote the control flow of a thread t crossing the boundary between the program and the TM: ***request*** actions correspond to the control being transferred from the former to the latter, and ***response*** actions, the other way around. A txbegin action is generated upon entering an `atomic` block, and a txcommit action when a transaction tries to commit upon exiting an `atomic` block. Actions call and ret denote a call to and a return from an invocation of a method on a transactional object and are annotated with the method parameter or return value. The TM may abort a transaction at any point when it is in control; this is recorded by an aborted response action.

A ***trace*** τ is a finite sequence of actions satisfying certain natural well-formedness conditions (stated informally due to space constraints; see Appendix B): every action in τ has a unique identifier; no action follows a `fault`; request and response actions are

properly matched; for every thread t, $\tau|_t$ cannot contain a request action immediately followed by a primitive action; actions denoting the beginning and end of transactions are properly matched; call and ret actions occur only inside transactions; and commands in τ do not access local variables of other threads and do not access global variables when inside a transaction. We denote the set of traces by Trace. A ***history*** is a trace containing only TM interface actions; we use H, S to range over histories. We specify the behavior of a TM implementation by the set of possible interactions it can have with programs: a ***transactional memory*** $\mathcal{T}$ is a set of histories that is prefix-closed and closed under renaming action identifiers.

We denote irrelevant expressions by _ and use the following notation: $\tau(i)$ is the i-th element of τ; $\tau|_t$ is the projection of τ onto actions of the form $(\_, t, \_)$; $|\tau|$ is the length of τ; $\tau_1\tau_2$ is the concatenation of τ_1 and τ_2. We say that an action φ is in τ, denoted by $\varphi \in \tau$, if $\tau = \_\varphi\_$. The empty sequence of actions is denoted ε.

A ***transaction*** T is a nonempty trace such that it contains actions by the same thread, begins with a txbegin action and only its last action can be a committed or an aborted action. We use the following terminology for transactions. A transaction T is: ***committed*** if it ends with a committed action, ***aborted*** if it ends with aborted, ***commit-pending*** if it ends with txcommit, and ***live***, in all other cases. We refer to this as T's ***status***. A transaction T is ***completed*** if it is either committed or aborted, and ***visible*** if it contains a txcommit action. A transaction T is ***in a trace*** τ, written $T \in \tau$, if $\tau|_t = \tau_1 T \tau_2$ for some t, τ_1 and τ_2, where either T is completed or τ_2 is empty. We denote the set of all transactions in τ by $\mathsf{tx}(\tau)$ and use self-explanatory notation for various subsets of transactions: $\mathsf{committed}(\tau)$, $\mathsf{aborted}(\tau)$, $\mathsf{pending}(\tau)$, $\mathsf{live}(\tau)$, $\mathsf{visible}(\tau)$. For $\varphi \in \tau$, the ***transaction of*** φ ***in*** τ, denoted $\mathsf{txof}(\varphi, \tau)$, is the subsequence of τ comprised of all actions that are in the same transaction in τ as φ (undefined if φ does not belong to a transaction).

4 Transactional Memory Specification (TMS)

In this section we define the TMS [5] correctness condition in our setting. TMS was originally formulated using I/O automata; here we define it in a different style appropriate for our goals (we provide further comparison in Section 7). Since threads may communicate through global variables outside of transactions, they may observe the *real-time order* between non-overlapping transactions in a history. Therefore, this order is a crucial building block in the TMS definition, as is common in consistency conditions for shared-memory concurrency, such as opacity [2] or linearizability [11].

Definition 2. *Let $\psi = (\_, t, \_)$ and $\psi' = (\_, t', \_)$ be two actions in a history H; ψ **is before** ψ' **in the real-time order** in H, denoted by $\psi \prec_H \psi'$, if $H = H\psi H_2 H_2' \psi' H_3$ and either (i) $t = t'$ or (ii) $(\_, t', \mathsf{txbegin}) \in H_2'\psi'$ and either $(\_, t, \mathsf{committed}) \in \psi H_2$ or $(\_, t, \mathsf{aborted}) \in \psi H_2$. A transaction T **is before** an action ψ' **in the real-time order** in H, denoted by $T \prec_H \psi'$, if $\psi \prec_H \psi'$ for every $\psi \in T$. A transaction T **is before** a transaction T' **in the real-time order** in H, denoted by $T \prec_H T'$, if $T \prec_H T'(1)$.*

The following *opacity relation* [2, 8] $H \sqsubseteq_{\mathsf{op}} S$ ensures that S is a a permutation of H preserving the real-time order.

Definition 3. *A history H is in the **opacity relation** with a history S, denoted by $H \sqsubseteq_{\mathsf{op}} S$, if $\forall \psi, \psi'. (\psi \in S \iff \psi \in H) \wedge (\psi \prec_H \psi' \implies \psi \prec_S \psi')$.*

Given a history H of program interactions with a concrete TM, TMS requires us to justify the behavior of all committed transactions in H by a single history S of the abstract TM, and to justify each response action ψ inside a transaction in H by an abstract history S_ψ. As we show in this paper, the existence of such justifications ensures that TMS implies observational refinement between the two TMs: the behavior of a program during some transaction in the history H of the program's interactions with the concrete TM can be reproduced when the program interacts with the abstract TM according to the history S or S_ψ. Below we use this insight when explaining the rationale for key TMS features.

The history S_ψ used to justify a response action ψ includes the transaction of ψ and a subset of transactions from H whose actions justify the response ψ. The following notion of a *possible past* of a history $H = H_1\psi$ defines all sets of transactions from H that can form S_ψ. Note that, if a transaction selected by this definition is aborted or commit-pending in H, its status is changed to committed when constructing S_ψ, as formalized later in Definition 5. Informally, the response ψ is given as if all the transactions in its possible past have taken effect and all the others have not. We first give the formal definition of a possible past, and then explain it using an example.

Definition 4. *A history $H_\psi = H_1'\psi$ is a **possible past** of a history $H = H_1\psi$, where ψ is a response action that it is not a* committed *or* aborted *action, if:*

(i) H_1' is a subsequence of H_1;

(ii) H_ψ is comprised of the transaction of ψ and some of the visible transactions in H: $\mathsf{tx}(H_\psi) \subseteq \{\mathsf{txof}(\psi, H)\} \cup \mathsf{visible}(H)$.

(iii) for every transaction $T \in H_\psi$, out of all transactions preceding T in the real-time order in H, the history H_ψ includes exactly the committed ones:

$$\forall T \in \mathsf{tx}(H_\psi).\, \forall T' \in \mathsf{tx}(H).\, T' \prec_H T \implies (T' \in \mathsf{tx}(H_\psi) \iff T' \in \mathsf{committed}(H)).$$

We denote the set of possible pasts of H by $\mathsf{TMSpast}(H)$.

We explain the definition using the history H of the trace shown in Figure 2; one of its possible pasts H_ψ consists of the transactions T_1, T_4 and T_5. According to (ii), the transaction of ψ (T_5 in Figure 2) is always included into any possible past, and live transactions are excluded: since they have not made an attempt to commit, they should not have an effect on ψ. Out of the visible transactions in H, we are allowed to select which ones to include (and, hence, treat as committed), subject to (iii): if we include a transaction T then, out of all transactions preceding T in the real-time order in H, we have to include exactly the committed ones. For example, since T_4 and T_5 are included in H_ψ, T_1 must also be included and T_3 must not. This condition is necessary for TMS to imply observational refinement. Informally, T_3 cannot be included into H_ψ because, in a program producing H, in between T_3 aborting and T_5 starting, thread t_2 could have communicated to thread t_3 the fact that T_3 has aborted, e.g., using a global variable g, as illustrated in Figure 2. When executing ψ, the code in T_5 may thus expect that T_3 did not take effect; hence, the result of ψ has to reflect this, so that the code behavior is preserved when replacing the concrete TM by an abstract one in observational refinement. This is a key idea used in our proof that TMS is necessary for observational refinement (Section 6.2). In contrast to T_3, we can include T_4 into H_ψ even if it is aborted or commit-pending. Since our language does not allow accessing

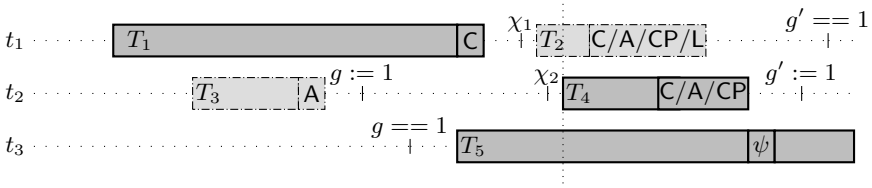

Fig. 2. Transactions T_1, T_4 and T_5 form one possible past of the history H of the trace shown. Allowed status of transactions in H is denoted as follows: committed – C, aborted – A, commit-pending – CP, live – L. The transaction T_5 executes only primitive actions after ψ in the trace.

global variables inside transactions, there is no way for the code in T_5 to find out about the status of T_4 from thread t_2, and hence, this code will not notice if the status of T_4 is changed to committed when replacing the concrete TM by an abstract one in observational refinement. For similar reasons, we can exclude T_2 from H_ψ even if it is committed. This idea is used in our proof that TMS is sufficient for observational refinement (Section 6.1).

Before giving the definition of TMS, we introduce operations used to change the status of transactions in a possible past of a history to committed. *Suffix commit completion* below converts commit-pending transactions into committed; then *completed possible past* defines a possible past with all transactions committed.

Definition 5. *A history H^c is a **suffix completion** of a history $H\psi$ if $H^c = H\psi H'$, every action in H' is either* committed *or* aborted*, and every transaction in H^c except possibly that of ψ, is completed. It is a **suffix commit completion** of H if H' consists of* committed *actions only. The sets of suffix completions and suffix commit completions of H are denoted* $\mathsf{comp}(H)$ *and* $\mathsf{ccomp}(H)$*, respectively.*

*A history H^c_ψ is a **completed possible past** of a history $H = H_1\psi$, if H^c_ψ is a suffix commit completion of a history obtained from a possible past $H'_1\psi$ of H by replacing all the* aborted *actions in H'_1 by* committed *actions. The set of completed possible pasts of H is denoted* $\mathsf{cTMSpast}(H)$*:*

$$\mathsf{cTMSpast}(H_1\psi) = \{H^c_\psi \mid \exists H'_1.\, H'_1\psi \in \mathsf{TMSpast}(H_1\psi) \wedge H^c_\psi \in \mathsf{ccomp}(\mathsf{com}(H'_1)\,\psi)\},$$

where $|\mathsf{com}(H'_1)| = |H'_1|$ *and*

$$\mathsf{com}(H'_1)(i) = (\text{if } (H'_1(i) = (a, t, \mathsf{aborted})) \text{ then } (a, t, \mathsf{committed}) \text{ else } H'_1(i)).$$

For example, one completed possible past of the history in Figure 2 consists of the transactions T_1, T_4 and T_5, with the status of the latter changed to committed if it was previously aborted or commit-pending. Note that a history H has a suffix completion only if H is of the form $H = H_1\psi$ where all the transactions in $H_1\psi$, except possibly that of ψ, are commit-pending or completed. Also, $\mathsf{cTMSpast}(H_1\psi) \neq \emptyset$ only if ψ is a response action.

The following definition of the *TMS relation* between TMs matches a history H arising from a concrete TM with a similar history S of an abstract TM. As part of this matching, we require that S preserves the real-time order of H. As in Definition 4(iii), this requirement is necessary to ensure observational refinement between the TMs: preserving the real-time order is necessary to preserve communication between threads when replacing the concrete TM with the abstract one.

Definition 6. *A history H is in the **TMS relation** with TM $\mathcal{T}$, denoted $H \sqsubseteq_{\mathsf{tms}} \mathcal{T}$, if:*

(i) $\exists H^c \in \mathsf{comp}(H|_{\neg\mathsf{live}}),\ S \in \mathcal{T}.\ H^c|_{\mathsf{com}} \sqsubseteq_{\mathsf{op}} S$*, where* $\cdot|_{\neg\mathsf{live}}$ *and* $\cdot|_{\mathsf{com}}$ *are the projections to actions by transactions that are not live and by committed transactions, respectively; and*

(ii) for every response action ψ such that it is not a committed *or* aborted *action and $H = H_1\psi H_2$, we have $\exists H^c_\psi \in \mathsf{cTMSpast}(H_1\psi).\ \exists S_\psi \in \mathcal{T}.\ H^c_\psi \sqsubseteq_{\mathsf{op}} S_\psi$.*

*A TM $\mathcal{T}_C$ is in the **TMS relation** with a TM $\mathcal{T}_A$, denoted by $\mathcal{T}_C \sqsubseteq_{\mathsf{tms}} \mathcal{T}_A$, if $\forall H \in \mathcal{T}_C.\ H \sqsubseteq_{\mathsf{tms}} \mathcal{T}_A$.*

5 Observational Refinement

Our main result relates TMS to *observational refinement*, which we introduce in this section. This requires defining the semantics of the programming language, i.e., the set of traces that computations of programs produce. Due to space constraints, we defer its formal definition to Appendix C and describe only its high-level structure. A ***state*** of a program records the values of all its variables: $s \in \mathsf{State} = \mathsf{Var} \to \mathbb{Z}$. The semantics of a program $P = C_1 \parallel \cdots \parallel C_m$ is given by the set of traces $[\![P, \mathcal{T}]\!](s) \subseteq \mathsf{Trace}$ it produces when executed with a TM $\mathcal{T}$ from an initial state s. To define this set, we first define the set of traces $[\![P]\!](s) \subseteq \mathsf{Trace}$ that a program can produce when executed from s with the behavior of the TM unrestricted, i.e., considering all possible values the TM can return to object method invocations and allowing transactions to commit or abort arbitrarily. We then restrict to the set of traces produced by P when executed with $\mathcal{T}$ by selecting those traces that interact with the TM in a way consistent with $\mathcal{T}$: $[\![P, \mathcal{T}]\!](s) = \{\tau \mid \tau \in [\![P]\!](s) \wedge \mathsf{history}(\tau) \in \mathcal{T}\}$, where $\mathsf{history}(\cdot)$ projects to TM interface actions. The definition of $[\![P]\!](s)$ follows the intuitive semantics of our programming language. In particular, it mandates that local variables be rolled back upon a transaction abort and includes traces corresponding to incomplete program computations into $[\![P]\!](s)$.

We can now define the notions of *observations* and *observational refinement*. Informally, given a trace τ of a client program, we consider observable: (i) the sequence of actions performed outside transactions in τ; (ii) the per-thread sequence of actions in τ excluding uncommitted transactions; and (iii) whether a τ ends with `fault` or not. Then observational refinement between a concrete TM $\mathcal{T}_C$ and an abstract one $\mathcal{T}_A$ states that every observable behavior of a program P using $\mathcal{T}_C$ can be reproduced when P uses $\mathcal{T}_A$. Hence, any conclusion about its observable behavior that a programmer makes assuming $\mathcal{T}_A$ will carry over to $\mathcal{T}_C$. Since our notion of observations excludes actions performed inside aborted or live transactions other than faulting, the programmer cannot make any conclusions about them. But, crucially, the programmer can be sure that, if a program is non-faulting under $\mathcal{T}_A$, it will stay so under $\mathcal{T}_C$. Let us call an action $\varphi \in \tau$ ***transactional*** if $\varphi \in T$ for some $T \in \tau$, and ***non-transactional*** otherwise. We denote by $\tau|_{\mathsf{trans}}$ and $\tau|_{\neg\mathsf{trans}}$ the projections of τ to transactional and non-transactional actions.

Definition 7. *The **thread-local observable behavior** of thread t in a trace τ, denoted by $\mathsf{observable}_t(\tau)$, is $\lightning$ if $\tau|_t$ ends with a* `fault` *action, and $(\tau|_t)|_{\mathsf{obs}}$ otherwise, where $\cdot|_{\mathsf{obs}}$ denotes the projection to non-transactional actions and actions by committed transactions. A TM $\mathcal{T}_C$ **observationally refines** a TM $\mathcal{T}_A$, denoted by $\mathcal{T}_C \preceq \mathcal{T}_A$, if for every program P, state s and trace $\tau \in [\![P, \mathcal{T}_C]\!](s)$ we have: (i) $\exists \tau' \in [\![P, \mathcal{T}_A]\!](s).\ \tau'|_{\neg\mathsf{trans}} = \tau|_{\neg\mathsf{trans}}$; and (ii) $\forall t.\ \exists \tau'_t \in [\![P, \mathcal{T}_A]\!](s).\ \mathsf{observable}_t(\tau'_t) = \mathsf{observable}_t(\tau)$.*

6 Main Result

The main result of this paper is that the TMS relation is equivalent to observational refinement for abstract TMs that enjoy certain natural closure properties. Their fomulation relies on the following notions.

A history H_a is an ***immediate abort extension of a history*** H if H is a subsequence of H_a, and whenever $\psi \in H_a$ and $\psi \notin H$ we have: (i) $\psi = (\_, \_, \mathsf{txbegin})$ or $\psi = (\_, \_, \mathsf{aborted})$, (ii) if $\psi = (\_, t, \mathsf{txbegin})$ then $H_a = H'_a \psi\, (\_, t, \mathsf{aborted})\, \_$, where $H'_a \in \{\varepsilon, \_\,(\_, \_, \mathsf{committed}), \_\,(\_, \_, \mathsf{aborted})\}$, and (iii) if $\psi = (\_, \_, \mathsf{aborted})$ then there exists $\psi' \notin H$ such that $H_a = \_\psi'\psi\_$. We denote by $\mathsf{addab}(H)$ the set of all immediate abort extensions of H. Informally, a history $H_a \in \mathsf{addab}(H)$ is an extension of H with transactions that abort immediately after their invocation. Note that the added transactions are placed either right before other transactions begin or right after they complete.

A history H_c is a ***non-interleaved completion*** of a history H if H is a subsequence of H_c, $\mathsf{pending}(H_c) = \emptyset$ and whenever $\psi \in H_c$ and $\psi \notin H$ we have $H_c = \_\,(\_, t, \mathsf{txcommit})\, \psi\_$ and either $\psi = (\_, t, \mathsf{committed})$ or $\psi = (\_, t, \mathsf{aborted})$. We denote the set of non-interleaved completions of H by $\mathsf{nicomp}(H)$. Informally, $H' \in \mathsf{nicomp}(H)$ completes each commit-pending transaction in H by adding a committed or aborted action at its end.

The required closure properties are formulated as follows:

CLP1 A TM $\mathcal{T}$ is ***closed under immediate aborts*** if whenever $H \in \mathcal{T}$ and $\mathsf{aborted}(H) = \emptyset$, we also have $H' \in \mathcal{T}$ for any history $H' \in \mathsf{addab}(H)$.

CLP2 A TM $\mathcal{T}$ is ***closed under removing transaction responses*** if whenever $H_1(\_, t, \mathsf{aborted})H_2 \in \mathcal{T}$ or $H_1(\_, t, \mathsf{committed})H_2 \in \mathcal{T}$ for H_2 not containing actions by t, we also have $H_1H_2 \in \mathcal{T}$.

CLP3 A TM $\mathcal{T}$ is ***closed under removing live and aborted transactions*** if whenever $H \in \mathcal{T}$, we also have $H' \in \mathcal{T}$ for any history H' which is a subsequence of H such that $\mathsf{committed}(H') = \mathsf{committed}(H)$, $\mathsf{pending}(H') = \mathsf{pending}(H)$, $\mathsf{live}(H') \subseteq \mathsf{live}(H)$ and $\mathsf{aborted}(H') \subseteq \mathsf{aborted}(H)$.

CLP4 A TM $\mathcal{T}$ is ***closed under completing commit-pending transactions*** if whenever $H \in \mathcal{T}$, we have $\mathsf{nicomp}(H) \cap \mathcal{T} \neq \emptyset$.

These properties are satisfied by the expected TM specification that executes every transaction atomically [8].

Theorem 1. *Let $\mathcal{T}_C$ and $\mathcal{T}_A$ be transactional memories.*
(i) If $\mathcal{T}_A$ satisfies CLP1 and CLP2, then $\mathcal{T}_C \sqsubseteq_{\mathsf{tms}} \mathcal{T}_A \Longrightarrow \mathcal{T}_C \preceq \mathcal{T}_A$.
(ii) If $\mathcal{T}_A$ satisfies CLP3 and CLP4, then $\mathcal{T}_C \preceq \mathcal{T}_A \Longrightarrow \mathcal{T}_C \sqsubseteq_{\mathsf{tms}} \mathcal{T}_A$.

6.1 Proof of Theorem 1(i) (Sufficiency)

Let us fix a program $P = C_1 \parallel \ldots \parallel C_m$ and a state s. As we have noted before, the main subtlety of TMS lies in justifying the behavior of a live transaction under $\mathcal{T}_C$ by a history of $\mathcal{T}_A$ where the committed/aborted status of some transactions is changed, as formalized by the use of $\mathsf{cTMSpast}$ in Definition 6(ii). Correspondingly, the most challenging part of the proof is to show that a trace from $[\![P, \mathcal{T}_C]\!](s)$ with a `fault` inside a live transaction can be transformed into a trace with the `fault` from $[\![P, \mathcal{T}_A]\!](s)$. The following lemma describes the first and foremost step of this transformation: given a

trace $\tau \in [\![P]\!](s)$ with a live transaction and a history $H^c_\psi \in \mathsf{cTMSpast}(\mathsf{history}(\tau))$, the lemma converts τ into another trace from $[\![P]\!](s)$ that contains the same live transaction, but whose history of non-aborted transactions is H^c_ψ. In other words, this establishes that the live transaction cannot notice changes in the committed/aborted status of other transactions done by $\mathsf{cTMSpast}$. Let $\tau|_{\neg\mathsf{abortedtx}}$ be the projection of τ excluding aborted transactions.

Lemma 1 (Live transaction insensitivity). *Let $\tau = \tau_1\psi\tau_2 \in [\![P]\!](s)$ be such that ψ is a response action by thread t_0 that is not a* committed *or* aborted *action and τ_2 is a sequence of primitive actions by thread t_0. For any $H^c_\psi \in \mathsf{cTMSpast}(\mathsf{history}(\tau))$ there exists $\tau_\psi \in [\![P]\!](s)$ such that $\mathsf{history}(\tau_\psi)|_{\neg\mathsf{abortedtx}} = H^c_\psi$ and $\tau_\psi|_{t_0} = \tau|_{t_0}$.*

Proof. We first show how to construct τ_ψ and then prove that it satisfies the required properties. We illustrate the idea of its construction using the trace τ in Figure 2. Let $\mathsf{history}(\tau) = H_1\psi$. Since $H^c_\psi \in \mathsf{cTMSpast}(H)$, by Definition 5 there exist histories H'_1, H''_1, and H^{cc} such that

$$H'_1\psi \in \mathsf{TMSpast}(H_1\psi) \ \wedge \ H''_1 = \mathsf{com}(H'_1) \ \wedge \ H^c_\psi = H''_1\psi H^{cc} \in \mathsf{ccomp}(H''_1\psi).$$

Recall that, for the τ in Figure 2, $H'_1\psi$ consists of the transactions T_1, T_4 and T_5. Then H''_1 is obtained from H'_1 by changing the last action of T_4 to committed if it was aborted; H^c_ψ is obtained by completing T_4 with a committed action if it was commit-pending. The trickiness of the proof comes from the fact that just mirroring these transformations on τ may not yield a trace of the program P: for example, if T_4 aborted, the code in thread t_2 following T_4 may rely on this fact, communicated to it by the TM via a local variable. Fortunately, we show that it is possible to construct the required trace by erasing certain suffixes of every thread and therefore getting rid of the actions that could be sensitive to the changes of transaction status, such as those following T_4. This erasure has to be performed carefully, since threads can communicate via global variables: for example, the value written by the assignment to g' in the code following T_4 may later be read by t_1, and, hence, when erasing the the former, the latter action has to be erased as well. We now explain how to truncate τ consistently.

Let ψ^b be the last txbegin action in $H'_1\psi$; then for some traces τ^b_1 and τ^b_2 we have $\tau = \tau^b_1\psi^b\tau^b_2\psi\tau_2$. For the τ in Figure 2, ψ^b is the txbegin action of T_4. Our idea is, for every thread other than t_0, to erase all its actions that follow the last of its transactions included into $H'_1\psi$ or its last non-transactional action preceding ψ^b, whichever is later. Formally, for every thread t, let τ^I_t denote the prefix of $\tau|_t$ that ends with the last TM interface action of t in $H'_1\psi$, or ε if no such action exists. For example, in Figure 2, $\tau^I_{t_1}$ and $\tau^I_{t_2}$ end with the last TM interface actions of T_1 and T_4, respectively. Similarly, let τ^N_t denote the prefix of $\tau|_t$ that ends in the last non-transactional action of t in τ^b_1, or ε if no such action exists. For example, in Figure 2, $\tau^N_{t_1}$ and $\tau^N_{t_2}$ end with χ_1 and χ_2, respectively. Let $\tau_{t_0} = \tau|_{t_0}$ and for each $t \neq t_0$ let τ_t be τ^I_t, if $|\tau^N_t| < |\tau^I_t|$, and τ^N_t, otherwise. We then let the truncated trace τ' be the subsequence of τ such that $\tau'|_t = \tau_t$ for each t. Thus, for the τ in Figure 2, in the corresponding trace τ' the actions of t_1 end with χ_1 and those of t_2 with the last action of T_4; note that this erases both operations on g'. To construct τ_ψ from τ', we mirror the transformations of H'_1 into H''_1 and H^c_ψ. Let τ'' be defined by $|\tau''| = |\tau'|$ and

$$\tau''(i) = (\text{if } (\tau'(i) = (a, t, \mathsf{aborted}) \wedge \tau'(i) \in H'_1)) \text{ then } (a, t, \mathsf{committed}) \text{ else } \tau'(i)).$$

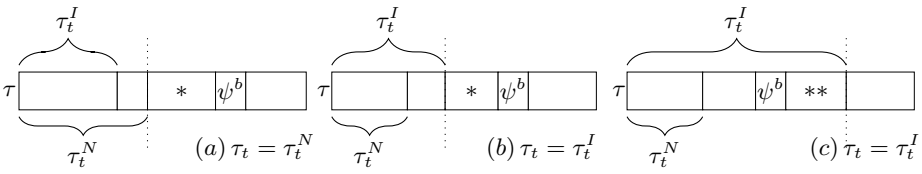

Fig. 3. Cases in the proof of Lemma 1. * all actions by t are transactional; ** all actions by t come from a single transaction, started before or by ψ^b

Then we let $\tau_\psi = \tau'' H^{cc}$.

We first prove that $\tau_\psi|_{t_0} = \tau|_{t_0}$. Let $T = \mathsf{txof}(\psi, H_1\psi)$; then by Definition 4(ii), $T \in H_1'\psi$. Hence, by Definition 4(iii) we have

$$\forall T'.\, T' \prec_{H_1'\psi} T \iff T' \prec_{H_1\psi} T \ \wedge\ T' \in \mathsf{committed}(H_1\psi), \tag{1}$$

so that $(H_1'\psi)|_{t_0}$ does not contain aborted transactions and $\tau''|_{t_0} = \tau'|_{t_0} = \tau|_{t_0}$. Besides, $H^{cc}|_{t_0} = \varepsilon$ and, hence, $\tau_\psi|_{t_0} = \tau''|_{t_0} = \tau|_{t_0}$.

We now sketch the proof that $\tau_\psi \in [\![P]\!](s)$, appealing to the intuitive understanding of the programming language semantics. To this end, we show that τ' and then τ'' belong to $[\![P]\!](s)$. We start by analyzing how the trace $\tau|_t$ is truncated to τ_t for every thread $t \neq t_0$. Let us make a case split on the relative positions of τ_t^N, τ_t^I and ψ^b in τ. There are three cases, shown in Figure 3. Either $\tau_t = \tau_t^N$ (a, thread t_1 in Figure 2) or $\tau_t = \tau_t^I$ (b, c). In the former case, ψ^b has to come after the end of τ_t^N. In the latter case, either ψ^b comes after the end of τ_t^I (b) or is its last action or precedes the latter (c, thread t_2 in Figure 2).

By the choice of τ_t^N, in (a) and (b) the fragment of τ in between the end of τ_t^N and ψ^b can contain only those actions by t that are transactional (T_2 in Figure 2). By the choice of τ_t^I and ψ^b, in (c) the fragment of τ in between ψ^b and the end of τ_t^I cannot contain a $\mathsf{txbegin}$ action by t; hence, by the choice of τ_t^N it can contain only those actions by t that are transactional. Furthermore, these have to come from a single transaction, started either by ψ^b or before it (T_4 in Figure 2). Finally, by the choice of ψ^b the actions of t_0 following ψ^b are transactional and come from the transaction of ψ, also started either by ψ^b or before it (T_5 in Figure 2). Given this analysis, the transformation from τ to τ' can be viewed as a sequence of two: (i) erase all actions following ψ^b, except those in some of transactions that were already ongoing at this time; (ii) erase some suffixes of threads containing only transactional actions. Since transactional actions do not access global variables, they are not affected by the actions of other threads. Furthermore, as we noted in Section 5, $[\![P]\!](s)$ includes incomplete program computations. This allows us to conclude that $\tau' \in [\![P]\!](s)$.

We now show that τ'' is valid, again referring to cases (a-c). Let $T = \mathsf{txof}(\psi^b, H_1\psi)$; then $T \in H_1'\psi$ by the choice of ψ^b and by Definition 4(iii) we get (1). Hence, for threads t falling into cases (a) or (b), $\tau'|_t$ does not contain aborted transactions that are also in $H_1'\psi$. For threads t falling into case (c), an aborted transaction by t included into $H_1'\psi$ can only be the last one in $\tau'|_t$. Finally, above we established that $(H_1'\psi)|_{t_0}$ does not contain aborted transactions. Hence, transactions in τ' whose status is changed from aborted to committed when switching to τ'' do not have any actions following them in τ'. Furthermore, $[\![P]\!](s)$ allows committing or aborting transactions

arbitrarily. This allows us to conclude that $\tau'' \in [\![P]\!](s)$. For the same reason, we get $\tau_\psi \in [\![P]\!](s)$.

Finally, we show that $\mathsf{history}(\tau_\psi)|_{\neg\mathsf{abortedtx}} = H^c_\psi$. It is sufficient to show that $\mathsf{history}(\tau'')|_{\neg\mathsf{abortedtx}} = H''_1\psi$; since $\tau_\psi = \tau''H^{cc}$ and H^{cc} contains only committed actions, this would imply

$$\mathsf{history}(\tau_\psi)|_{\neg\mathsf{abortedtx}} = \mathsf{history}(\tau''H^{cc})|_{\neg\mathsf{abortedtx}} = \mathsf{history}(\tau'')|_{\neg\mathsf{abortedtx}}H^{cc} = H''_1\psi H^{cc} = H^c_\psi.$$

By the choice of τ^I_t for $t \neq t_0$, every transaction in $(H'_1\psi)|_t$ is also in τ^I_t. Hence, $H'_1\psi$ is a subsequence of $\mathsf{history}(\tau')$. By the definition of τ'' and H''_1, $H''_1\psi$ is a subsequence of $\mathsf{history}(\tau'')$. Then since $H''_1\psi$ does not contain aborted transactions, $H''_1\psi$ is a subsequence of $\mathsf{history}(\tau'')|_{\neg\mathsf{abortedtx}}$.

Thus, to prove $\mathsf{history}(\tau'')|_{\neg\mathsf{abortedtx}} = H''_1\psi$ it remains to show that every non-aborted transaction in $\mathsf{history}(\tau'')$ is in $H''_1\psi$. Since the construction of τ'' from τ' changes the status of only those transactions that belong to $H'_1\psi$, it is sufficient to show that every non-aborted transaction in $\mathsf{history}(\tau')$ is in $H'_1\psi$. Here we only consider the case when such a transaction is by a thread $t \neq t_0$ and $\tau'|_t = \tau^N_t \neq \varepsilon$; we cover the other cases in Appendix D. Let χ^N_t be the last action in τ^N_t and $T = \mathsf{txof}(\psi^b, H_1\psi) \in H'_1\psi$. Then by Definition 4(iii) we get (1). Since χ^N_t comes before ψ^b in $H_1\psi$, any transaction T' in $\tau'|_t$ is such that $T' \prec_{H_1\psi} T$, which together with (1) implies the required. This concludes the proof that $\mathsf{history}(\tau'')|_{\neg\mathsf{abortedtx}} = H''_1\psi$. □

We now give the other lemmas necessary for the proof. Definition 6 matches a history of $\mathcal{T}_C$ with one of $\mathcal{T}_A$ using the opacity relation, possibly after transforming the former with cTMSpast. The following lemma is used to transform a trace of P accordingly. The lemma shows that, if we consider only traces where aborted transactions abort immediately (i.e., are of the form $(\_,\_,\mathsf{txbegin})\,(\_,\_,\mathsf{aborted})$) , then the opacity relation implies observational refinement with respect to observing non-transactional actions and thread-local trace projections. This result is a simple adjustment of the one about the sufficiency of opacity for observational refinement to our setting [8, Theorem 16] (it was proved in [8] for a language where local variables are *not* rolled back upon a transaction abort; this difference, however, does not matter if aborted transactions abort immediately).

Lemma 2. *Consider $\tau \in [\![P]\!](s)$ such that all the aborted transactions in τ abort immediately. Let S be such that $\mathsf{history}(\tau) \sqsubseteq_{\mathsf{op}} S$. Then there exists $\tau' \in [\![P]\!](s)$ such that $\mathsf{history}(\tau') = S$, $\tau|_{\neg\mathsf{trans}} = \tau'|_{\neg\mathsf{trans}}$ and $\forall t.\, \tau'|_t = \tau|_t$.*

Let $\tau|_{\neg\mathsf{abortact}}$ be the trace obtained from τ by removing all actions inside aborted transactions, so that every such transaction aborts immediately. We can benefit from Lemma 2 because local variables are rolled back if a transaction aborts, and, hence, applying $\cdot|_{\neg\mathsf{abortact}}$ to a trace preserves its validity.

Proposition 1. $\forall\tau.\, \tau \in [\![P]\!](s) \implies \tau|_{\neg\mathsf{abortact}} \in [\![P]\!](s)$.

Finally, Definition 6 matches only histories of committed transactions, but the histories of the traces in Lemma 2 also contain aborted transactions. Fortunately, the following lemma allows us to add empty aborted transactions into the abstract history while preserving the opacity relation.

Lemma 3. *Let H be a history where all aborted transactions abort immediately and S be such that $H|_{\neg\mathsf{abortedtx}} \sqsubseteq_{\mathsf{op}} S$. There exists a history $S' \in \mathsf{addab}(S)$ such that $H \sqsubseteq_{\mathsf{op}} S'$.*

Definition 6(i), Proposition 1 and Lemmas 2 and 3 can be used to prove that the TMS relation preserves non-transactional actions and thread-local observable behavior of threads whose last action is not a `fault`.

Lemma 4. *If $\mathcal{T}_C \sqsubseteq_{\mathsf{tms}} \mathcal{T}_A$ and $\mathcal{T}_A$ satisfies CLP1 and CLP2, then*

$$\forall \tau \in [\![P, \mathcal{T}_C]\!](s).\, \exists \tau' \in [\![P, \mathcal{T}_A]\!](s).\, (\tau'|_{\neg\mathsf{trans}} = \tau|_{\neg\mathsf{trans}}) \wedge (\forall t.\, (\tau'|_t)|_{\mathsf{obs}} = (\tau|_t)|_{\mathsf{obs}}).$$

Proof of Theorem 1(i). Given Lemma 4, we only need to establish the preservation of faults inside transactions. Consider $\tau_0 \in [\![P, \mathcal{T}_C]\!](s)$ such that $\tau_0 = \tau_1 \psi \tau_2 \chi$, where $\chi = (\_, t_0, \texttt{fault})$ is transactional and ψ is the last TM interface action by thread t_0. Then $\tau_2|_{t_0}$ consists of transactional actions and thus does not contain accesses to global variables. Hence, $\tau = \tau_1 \psi (\tau_2|_{t_0}) \chi \in [\![P, \mathcal{T}_C]\!](s)$. By our assumption, $\mathcal{T}_C \sqsubseteq_{\mathsf{tms}} \mathcal{T}_A$. Then there exists $H^c_\psi \in \mathsf{cTMSpast}(\mathsf{history}(\tau))$ and $S \in \mathcal{T}_A$ such that $H^c_\psi \sqsubseteq_{\mathsf{op}} S$. By Lemma 1, for some trace τ_ψ we have $\tau_\psi \in [\![P]\!](s)$, $\mathsf{history}(\tau_\psi)|_{\neg\mathsf{abortedtx}} = H^c_\psi$ and $\tau_\psi|_{t_0} = \tau|_{t_0}$. By Proposition 1, $\tau_\psi|_{\neg\mathsf{abortact}} \in [\![P]\!](s)$. Using Lemma 3, we get a history S' such that $\mathsf{history}(\tau_\psi|_{\neg\mathsf{abortact}}) \sqsubseteq_{\mathsf{op}} S'$ and $S' \in \mathsf{addab}(S)$. Since $S \in \mathcal{T}_A$ and $\mathcal{T}_A$ is closed under immediate aborts (CLP1), we get $S' \in \mathcal{T}_A$. Hence, by Lemma 2, for some $\tau' \in [\![P, \mathcal{T}_A]\!](s)$ we have $\tau'|_{t_0} = \tau_\psi|_{t_0} = \tau|_{t_0} = \_\chi$, as required. □

6.2 Proof Sketch for Theorem 1(ii) (Necessity)

Consider $\mathcal{T}_C$ and $\mathcal{T}_A$ such that $\mathcal{T}_C \preceq \mathcal{T}_A$ and $\mathcal{T}_A$ satisfies the closure conditions stated in the theorem. To show that for any $H_0 \in \mathcal{T}_C$ we have $H_0 \sqsubseteq_{\mathsf{tms}} \mathcal{T}_A$, we have to establish conditions (i) and (ii) from Definition 6. We sketch the more interesting case of (ii), in which $H_0 = H_1 \psi H_2 = H H_2 \in \mathcal{T}_C$, where ψ is a response action by a thread t_0 that is not a committed or aborted action. We need to find $H^c \in \mathsf{cTMSpast}(H)$ and $S \in \mathcal{T}_A$ such that $H^c \sqsubseteq_{\mathsf{op}} S$.

To this end, we construct a program P_H (as we explain further below) where every thread t performs the sequence of transactions specified in $H|_t$. The program monitors certain properties of the TM behavior, e.g., checking that the return values obtained from methods of transactional objects in committed transactions correspond to those in H and that the real-time order between actions includes that in H. If these properties hold, thread t_0 ends by executing the `fault` command. Let s be a state with all variables set to distinguished values. We next construct a trace $\tau \in [\![P_H, \mathcal{T}_C]\!](s)$ such that $\mathsf{history}(\tau) = H$ and t_0 faults in τ. By Definition 7, there exists $\tau' \in [\![P_H, \mathcal{T}_A]\!](s)$ such that t_0 faults in τ'. However, the program P_H is constructed so that t_0 can fault in τ' only if the properties of the TM behaviour the program monitors hold, and thus H is related to $\mathsf{history}(\tau')$ in a certain way. This relationship allows us to construct $H^c \in \mathsf{cTMSpast}(H)$ from H and $S \in \mathcal{T}_A$ from $\mathsf{history}(\tau')$ such that $H^c \sqsubseteq_{\mathsf{op}} S$.

In more detail, thread t_0 in P_H monitors the return status of every transaction and the return values obtained inside the atomic blocks corresponding to transactions committed in $H|_{t_0}$ and the (live) transaction of ψ. If there is a mismatch with $H|_{t_0}$, this is recorded in a special local variable. At the end of the transaction of ψ, t_0 checks the variable and faults if the TM behavior matched $H|_{t_0}$. This construction is motivated by

the fact that faulting is the only observation Definition 7 allows us to make about the behavior of the live transaction of ψ. Since the definition does not correlate actions by threads t other than t_0 between τ and τ', such threads monitor TM behavior differently: if there is a mismatch with $H|_t$, a thread t faults immediately. Since a trace can have at most one `fault` and t_0 faults in τ', this ensures that any committed transaction in τ' behaves as in H.

To check whether an execution of P_H complies with the real-time order in H, for each transaction in H, we introduce a global variable g, which is initially 0 and is set to 1 by the thread executing the transaction right after the transaction completes, by a command following the corresponding atomic block. Before starting a transaction, each thread checks whether all transactions preceding this one in the real-time order in H have finished by reading the corresponding g variables. Thread t_0 records the outcome in the special local variable checked at the end; all other threads fault upon detecting a mismatch.

Let $H' = \mathsf{history}(\tau')$. This construction of P_H allows us to infer that: (i) the projection of $H'|_{t_0}$ to committed transactions and $\mathsf{txof}(\psi, H')$ is equal to the corresponding projection of $H|_{t_0}$; (ii) for all other threads t a similar relationship holds for the prefix of $H'|_t$ ending with the last transaction preceding $\mathsf{txof}(\psi, H')$ in the real-time order; (iii) the real-time order in H' includes that in H. Transactions concurrent with $\mathsf{txof}(\psi, H')$ in H' may behave differently from H. However, checks done by P_H inside these transactions ensure that, if such a transaction T is visible in H', then the return values inside T match those in H. The checks on the global variables g done right before T also ensure that all transactions preceding T in the real-time order in H commit or abort in H' as prescribed by H. This relationship between H and H' allows us to establish the requirements of Definition 6(ii). □

7 Related Work

When presenting TMS [5], 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 Definition 6(i) ensures the correctness of committed transactions. The discussion of the most challenging case of live transactions—corresponding to Definition 6(ii) and our Lemma 1—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 Definition 6(ii) ensure this. This reasoning is very delicate, as indicated by our proof of Lemma 1, 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*) [3]. Like TMS, VWC allows every operation in a live or aborted transaction to be justified 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 [8] 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 developed new techniques, such as establishing the live transaction insensitivity property (Lemma 1) to prove sufficiency and proposing monitor programs for the nontrivial constraints used in the TMS definition to prove necessity. Similarly to [8] and other papers using observational refinement to study consistency conditions [12, 13], we reformulate TMS so that it is not restricted to a particular abstract TM $\mathcal{T}_A$. This generality, not allowed by the original TMS definition, has two benefits. First, our reformulation can be used to compare two TM implementations, e.g., an optimized and an unoptimized one. Second, dealing with the general definition forces us to explicitly state the closure properties required from the abstract TM, rather than having them follow implicitly from its atomic behavior.

Acknowledgements. We thank Mohsen Lesani, Victor Luchangco and the anonymous reviewers for comments that helped us improve the paper. This work was supported by EU FP7 project ADVENT (308830).

References

1. Papadimitriou, C.H.: The serializability of concurrent database updates. J. ACM **26** (1979) 631–653
2. Guerraoui, R., Kapalka, M.: On the correctness of transactional memory. In: PPOPP. (2008) 175–184
3. Imbs, D., Raynal, M.: Virtual world consistency: A condition for STM systems (with a versatile protocol with invisible read operations). Theor. Comput. Sci. **444** (2012) 113–127
4. Attiya, H., Hans, S., Kuznetsov, P., Ravi, S.: Safety of deferred update in transactional memory. In: ICDCS. (2013)
5. Doherty, S., Groves, L., Luchangco, V., Moir, M.: Towards formally specifying and verifying transactional memory. Formal Aspects of Computing **25** (2013) 769–799
6. He, J., Hoare, C., Sanders, J.: Prespecification in data refinement. Information Processing Letters **25** (1987) 71 – 76
7. He, J., Hoare, C., Sanders, J.: Data refinement refined. In: ESOP. (1986) 187–196
8. Attiya, H., Gotsman, A., Hans, S., Rinetzky, N.: A programming language perspective on transactional memory consistency. In: PODC. (2013) 309–318
9. Scala STM Expert Group: Scala STM quick start guide (2012) http://nbronson.github.io/scala-stm/quick_start.html.
10. Lesani, M., Luchangco, V., Moir, M.: Putting opacity in its place. In: WTTM. (2012)
11. Herlihy, M., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Transactions on Programming Languages and Systems **12** (1990) 463–492
12. Gotsman, A., Yang, H.: Liveness-preserving atomicity abstraction. In: ICALP (2). (2011) 453–465
13. Filipovic, I., O'Hearn, P.W., Rinetzky, N., Yang, H.: Abstraction for concurrent objects. In: ESOP. (2009) 252–266

A Restrictions on Variable Accesses

To formalise restrictions on accesses to variables by primitive commands, we partition the set $\mathsf{PComm} - \{\texttt{fault}\}$ into $2m$ classes: $\mathsf{PComm} - \{\texttt{fault}\} = \biguplus_{t=1}^{m}(\mathsf{LPcomm}_t \uplus \mathsf{GPcomm}_t)$. The intention is that commands from LPcomm_t can access only the local variables of thread t (LVar_t); commands from GPcomm_t can additionally access global variables ($\mathsf{LVar}_t \uplus \mathsf{GVar}$). We formalize these restrictions in Appendix C. To ensure that a thread t does not access local variables of other threads, we require that the thread cannot mention such variables in the conditions of `if` and `while` commands and can only use primitive commands from $\mathsf{LPcomm}_t \uplus \mathsf{GPcomm}_t$.

B Formal Definition of Traces

Definition 8 (Traces). *A **trace** τ is a finite sequence of actions, satisfying the following conditions:*

(i) every action in τ has a unique identifier: if $\tau = \tau_1(a_1, \_, \_)\tau_2(a_2, \_, \_)\tau_3$ then $a_1 \neq a_2$.

(ii) no action follows a `fault`*: if $\tau = \tau'\varphi$ then τ' does not contain a* `fault` *action.*

(iii) request and response actions are properly matched: for every thread t, $\mathsf{history}(\tau)|_t$ consists of alternating request and corresponding response actions, starting from a request action;

(iv) for every thread t, $\tau|_t$ cannot contain a request action immediately followed by a primitive action;

(v) actions denoting the beginning and end of transactions are properly matched: for every thread t, in the projection of $\tau|_t$ to txbegin*,* committed *and* aborted *actions,* txbegin *alternates with* committed *or* aborted*, starting from* txbegin*;*

(vi) call *and* ret *actions occur only inside transactions: for every thread t, if $\tau|_t = \tau_1\psi\tau_2$ for a* call *or* ret *action ψ, then $\tau_1 = \tau_1'\psi'\tau_1''$ for some* txbegin *action ψ', and τ_1' and τ_1'' such that τ_1'' does not contain* committed *or* aborted *actions;*

(vii) commands in τ do not access local variables of other threads: if $(\_, t, c) \in \tau$ then $c \in \mathsf{LPcomm}_t \uplus \mathsf{GPcomm}_t \uplus \{\texttt{fault}\}$;

(viii) commands in τ do not access global variables inside a transaction: if $\tau = \tau_1\,(\_, t, c)\,\tau_2$ for $c \in \mathsf{GPcomm}_t$, then it is not the case that $\tau_1 = \tau_1'\,(\_, t, \mathsf{txbegin})\,\tau_1''$, where τ_1'' does not contain committed *or* aborted *actions.*

C Formal Definition of the Semantics of the Programming Language

This section formally defines the set $[\![P]\!](s)$. It is computed in two stages. First, we compute a set $A(P)$ of traces that resolves all issues regarding sequential control flow and interleaving. Intuitively, if one thinks of each thread C_t in P as a control-flow graph, then $A(P)$ contains all possible interleavings of paths in the graphs of C_t, $t \in \mathsf{ThreadID}$ starting from their initial nodes. The set $A(P)$ is a superset of all the traces that can actually be executed: e.g., if a thread executes the command "$x := 1$; `if` $(x = 1)$ $y := 1$ `else` $y := 2$" where x, y are local variables, then $A(P)$ will contain a trace where $y := 2$ is executed instead of $y := 1$. To filter out such nonsensical traces, we

$$A'(c)t = \{(\_, t, c)\}$$

$$A'(C_1; C_2)t = \{\tau_1\,\tau_2 \mid \tau_1 \in A'(C_1)t \wedge \tau_2 \in A'(C_2)t\}$$

$$A'(\texttt{if}\ (b)\ \texttt{then}\ C_1\ \texttt{else}\ C_2)t = \{(\_, t, \mathsf{assume}(b))\,\tau_1 \mid \tau_1 \in A'(C_1)t\} \cup \{(\_, t, \mathsf{assume}(\neg b))\,\tau_2 \mid \tau_2 \in A'(C_2)t\}$$

$$A'(\texttt{while}\ (b)\ \texttt{do}\ C)t = \{((\_, t, \mathsf{assume}(b))\,(A'(C)t))^*\,(\_, t, \mathsf{assume}(\neg b))\}$$

$$A'(x := o.f(e))t = \{(\_, t, \mathsf{assume}(e = n))\,(\_, t, \mathsf{call}\ o.f(n))\,(\_, t, \mathsf{ret}(n')\ o.f)\,(\_, t, x := n') \mid n, n' \in \mathbb{Z}\} \cup \{(\_, t, \mathsf{assume}(e = n))\,(\_, t, \mathsf{call}\ o.f(n))\,(\_, t, \mathsf{aborted}) \mid n \in \mathbb{Z}\}$$

$$A'(x := \texttt{atomic}\,\{C\})t = \{(\_, t, \mathsf{txbegin})\,(\_, t, \mathsf{aborted})\,(\_, t, x := \mathsf{aborted})\} \cup \{(\_, t, \mathsf{txbegin})\,(\_, t, \mathsf{OK})\,\tau\,(\_, t, \mathsf{aborted})\,(\_, t, x := \mathsf{aborted}) \mid \tau\,(\_, t, \mathsf{aborted})\,\tau' \in A'(C)t \wedge (\_, t, \mathsf{aborted}) \notin \tau\} \cup \{(\_, t, \mathsf{txbegin})\,(\_, t, \mathsf{OK})\,\tau\,(\_, t, \mathsf{txcommit})\,(\_, t, r)\,(\_, t, x := r) \mid \tau \in A'(C)t \wedge (\_, t, \mathsf{aborted}) \notin \tau \wedge (r = \mathsf{committed} \vee r = \mathsf{aborted})\}$$

$$A'(C_1 \parallel \ldots \parallel C_m) = \mathsf{prefix}(\bigcup\{\mathsf{interleave}(\tau_1, \ldots, \tau_m) \mid \forall t.\, 1 \le t \le m \implies \tau_t \in A'(C_t)t\})$$

$$A(P) = A'(P) \cap \mathsf{Trace}$$

Fig. 4. The definition of $A(P)$.

evaluate every trace to determine whether it is ***valid***, i.e., whether its control flow is consistent with the effect of its actions on program variables. This is formalized by a function $\mathsf{eval} : \mathsf{State} \times \mathsf{Trace} \to \mathcal{P}(\mathsf{State}) \cup \{\lightning\}$ that, given an initial state and a trace, produces the set of states resulting from executing the actions in the trace, an empty set if the trace is invalid, or a special state $\lightning$ if the trace contains a `fault` action. Thus, $[\![P]\!](s) = \{\tau \in A(P) \mid \mathsf{eval}(s, \tau) \neq \emptyset\}$.

When defining the semantics, we encode the evaluation of conditions in `if` and `while` statements with assume commands. More specifically, we expect that the sets LPcomm_t contain special primitive commands $\mathsf{assume}(b)$, where b is a Boolean expression over local variables of thread t, defining the condition. We state their semantics formally below; informally, $\mathsf{assume}(b)$ does nothing if b holds in the current program state, and stops the computation otherwise. Thus, it allows the computation to proceed only if b holds. The assume commands are only used in defining the semantics of the programming language; hence, we forbid threads from using them directly.

The trace set $A(P)$. The function $A'(\cdot)$ in Figure 4 maps commands and programs to sequences of actions they may produce. Technically, $A'(\cdot)$ might contain sequences that are not traces, e.g., because they do not have unique identifiers or continue beyond a `fault` command. This is resolved by intersecting the set $A'(P)$ with the set of all traces to define $A(P)$. $A'(C)t$ gives the set of action sequences produced by a command C when it is executed by thread t. To define $A'(P)$, we first compute the set of all the interleavings of action sequences produced by the threads constituting P. Formally, $\tau \in \mathsf{interleave}(\tau_1, \ldots, \tau_m)$ if and only if every action in τ is performed by some thread $t \in \{1, \ldots, m\}$, and $\tau|_t = \tau_t$ for every thread $t \in \{1, \ldots, m\}$. We then let $A'(P)$ be the set of all prefixes of the resulting sequences, as denoted by the prefix operator.

We take prefix closure here to account for incomplete program computations as well as those in which the scheduler preempts a thread forever.

$A'(c)t$ returns a singleton set with the action corresponding to the primitive command c (primitive commands execute atomically). $A'(C_1; C_2)t$ concatenates all possible action sequences corresponding to C_1 with those corresponding to C_2. The set of action sequences of a conditional considers cases where either branch is taken. We record the decision using an assume action; at the evaluation stage, this allows us to ensure that this decision is consistent with the program state. The set of action sequences for a loop is defined using the Kleene closure operator * to produce all possible unfoldings of the loop body. Again, we record branching decisions using assume actions.

The set of action sequences of a method invocation $x := f(e)$ includes both sequences where the method executes successfully and where the current transaction is aborted. The former set is constructed by nondeterministically choosing two integers n and n' to describe the parameter n and the return value n' for the method call. To ensure that e indeed evaluates to n, we insert $\mathsf{assume}(e = n)$ before the call action, and to ensure that x gets the return value n', we add the assignment $x := n'$ after the ret action. Note that some of the choices here might not be feasible: the chosen n might not be the value of the parameter expression e when the method is invoked, or the method might never return n' when called with n. Such infeasible choices are filtered out at the following stages of the semantics definition: the former in the definition of $[\![P]\!](s)$ by the use of evaluation and the semantics of assume, and the latter in the definition of $[\![P, \mathcal{T}]\!](s)$ by selecting the sequences from $[\![P]\!](s)$ that interact with the transactional memory correctly. The set of action sequences of $x := \texttt{atomic}\,\{C\}$ contains those in which C is aborted in the middle of its execution (at an object operation or right after it begins) and those in which C executes until completion and then the transaction commits or aborts.

Semantics of primitive commands. To define evaluation, we assume a semantics of every command $c \in \mathsf{PComm} - \{\texttt{fault}\}$, given by a function $[\![c]\!]$ that defines how the program state is transformed by executing c. As we noted before, different classes of primitive commands are supposed to access only certain subsets of variables. To ensure that this is indeed the case, we define $[\![c]\!]$ as a function of only those variables that c is allowed to access. Namely, the semantics of $c \in \mathsf{LPcomm}_t$ is given by

$$[\![c]\!] : (\mathsf{LVar}_t \to \mathbb{Z}) \to \mathcal{P}(\mathsf{LVar}_t \to \mathbb{Z}).$$

The semantics of $c \in \mathsf{GPcomm}_t$ is given by

$$[\![c]\!] : ((\mathsf{LVar}_t \uplus \mathsf{GVar}) \to \mathbb{Z}) \to \mathcal{P}((\mathsf{LVar}_t \uplus \mathsf{GVar}) \to \mathbb{Z}).$$

Note that we allow c to be non-deterministic.

For a valuation q of variables that c is allowed to access, $[\![c]\!](q)$ yields the set of their valuations that can be obtained by executing c from a state with variable values q. For example, an assignment command $x := g$ has the following semantics:

$$[\![x := g]\!](q) = \{q[g \mapsto q(g)]\}\ .$$

We define the semantics of assume commands following the informal explanation given at the beginning of this section: for example,

$$[\![\mathsf{assume}(x = n)]\!](q) = \begin{cases} \{q\}, & \text{if } q(x) = n; \\ \emptyset, & \text{otherwise.} \end{cases} \tag{2}$$

Thus, when the condition in assume does not hold of q, the command stops the computation by not producing any output.

We lift functions $[\![c]\!]$ to full states by keeping the variables that c is not allowed to access unmodified and producing $\lightning$ if c faults. For example, if $c \in \mathsf{LPcomm}_t$, then

$$[\![c]\!](s) = \{s|_{\mathsf{LVar}\setminus\mathsf{LVar}_t} \uplus q \mid q \in [\![c]\!](s|_{\mathsf{LVar}_t})\},$$

where $s|_V$ is the restriction of s to variables in V. Finally, we let

$$[\![\texttt{fault}]\!](s) = \lightning,$$

so that the only way a program can fault is by executing the `fault` command.

Trace evaluation. Using the semantics of primitive commands, we first define the evaluation of a single action on a given state:

$$\begin{aligned} &\mathsf{eval} : \mathsf{State} \times \mathsf{Action} \to \mathcal{P}(\mathsf{State}) \cup \{\lightning\} \\ &\mathsf{eval}(s, (\_, t, c)) \;=\; [\![c]\!](s); \\ &\qquad \mathsf{eval}(s, \psi) \;=\; \{s\}. \end{aligned}$$

Note that this does not change the state s as a result of TM interface actions, since their return values are assigned to local variables by separate actions introduced when generating $A(P)$. We then lift eval to traces as follows:

$$\begin{aligned} &\mathsf{eval} : \mathsf{State} \times \mathsf{Trace} \to \mathcal{P}(\mathsf{State}) \cup \{\lightning\} \\ &\mathsf{eval}(s, \tau) = \begin{cases} \emptyset, & \text{if } \tau = \tau'\varphi \wedge \mathsf{eval}(s, \tau') = \emptyset; \\ \mathsf{evalna}(s, \tau|_{\neg\mathsf{abortact}}), & \text{otherwise,} \end{cases} \end{aligned}$$

where

$$\mathsf{evalna}(s, \tau) = \begin{cases} \{s\}, & \text{if } \tau = \varepsilon; \\ \{s'' \in \mathsf{eval}(s', \varphi) \mid s' \in \mathsf{evalna}(s, \tau')\}, & \text{if } \tau = \tau'\varphi. \end{cases}$$

The set of states resulting from evaluating trace τ from state s is effectively computed by the helper function $\mathsf{evalna}(s, \tau)$, which ignores actions inside aborted transactions to model local variable roll-back. However, ignoring the contents of aborted transactions completely poses a risk that we might consider traces including sequences of actions inside aborted transactions that yield an empty set of states. To mitigate this, $\mathsf{eval}(s, \tau)$ recursively evaluates every prefix of τ, thus ensuring that sequences of actions inside aborted transaction are valid.

As we explained in Section 5, we define $[\![P]\!](s)$ as the set of those traces from $A(P)$ that can be evaluated from s without getting stuck, as formalized by eval. Note that this

definition enables the semantics of assume defined by (2) to filter out traces that make branching decisions inconsistent with the program state. For example, consider again the program "$x := 1;\ \texttt{if}\ (x = 1)\ y := 1\ \texttt{else}\ y := 2$". The set $A(P)$ includes traces where both branches are explored. However, due to the semantics of the assume actions added to the traces according to Figure 4, only the trace executing $y := 1$ will result in a nonempty set of final states after the evaluation and, therefore, only this trace will be included into $[\![P]\!](s)$.

D Additional Proofs

D.1 Remaining Cases from the Proof of Lemma 1

- $t \neq t_0$ is such that $\tau'|_t = \tau_t^I \neq \varepsilon$. Let ψ_t^I be the last action in τ_t^I. Let $T = \mathsf{txof}(\psi_t^I, H_1\psi)$. By the choice of τ_t^I we have $T \in H_1'\psi$; then by Definition 4(iii) we get (1). Since any transaction T' in $\mathsf{history}(\tau'|_t)$ is either T or is such that $T' \prec_{(H_1\psi)|_t} T$, this implies the required.
- $t = t_0$. Let $T = \mathsf{txof}(\psi, H_1\psi) \in H_1'\psi$. Then by Definition 4(iii) we get (1). Since any transaction T' in $\mathsf{history}(\tau'|_{t_0})$ is either T or is such that $T' \prec_{(H_1\psi)|_{t_0}} T$, this implies the required.

D.2 Proof of Lemma 3

Let n be the number of aborted transactions in H. To construct the desired S', we inductively construct a sequence of histories S_i, $i = 0..n$ such that

$$|\mathsf{aborted}(S_i)| = i; \qquad S_i \in \mathsf{addab}(S); \qquad \{\psi \mid \psi \in S_i\} \subseteq \{\psi \mid \psi \in H\}; \\ \forall \psi_1, \psi_2 \in S_i.\, \psi_1 \prec_H \psi_2 \implies \psi_1 \prec_{S_i} \psi_2. \tag{3}$$

We then let $S' = S_n$, so that $H \sqsubseteq_{\mathsf{op}} S'$.

For $i = 0$, we take $S_0 = S$, and all the requirements in (3) hold vacuously. Assume a history S_i satisfying (3) was constructed; we get S_{i+1} from S_i by the following construction. Let $H = H_1\psi_b H_2 \psi_a H_3$, where $\psi_b = (\_, t, \mathsf{txbegin}), \psi_a = (\_, t, \mathsf{aborted})$, $H_2|_t = \varepsilon$ and

$$\neg\exists \psi'.\, \psi' = (\_, \_, \mathsf{txbegin}) \in H_1 \wedge \mathsf{txof}(\psi', H) \in \mathsf{aborted}(H) \wedge \psi' \notin S_i.$$

That is, out of all aborted transactions in H that are not in S_i, $\psi_b\psi_a$ is the one with the earliest txbegin. We now consider two cases.

Case I: H_1 does not contain a committed or an aborted action.

In this case, let $S_{i+1} = \psi_b\psi_a S_i$. We only need to show that for any $\psi' \in S_i$ we have $\psi' \prec_H \psi_b \implies \psi' \prec_{S_{i+1}} \psi_b$ and $\psi_a \prec_H \psi' \implies \psi_a \prec_{S_{i+1}} \psi'$. The latter holds by the construction of S_{i+1}. To show the former, observe that, since H_1 does not contain a committed or aborted action, it cannot contain actions by thread t. Hence, we cannot have $\psi' \prec_H \psi_b$ for any ψ'.

Case II: H_1 contains a committed or an aborted action.

Let ψ be the last committed or aborted action in S_i that is also in H_1 and let $S_i = S'\psi S''$. We then let $S_{i+1} = S'\psi\psi_b\psi_a S''$. We again need to show that for any $\psi' \in S_i$ we have $\psi' \prec_H \psi_b \implies \psi' \prec_{S_{i+1}} \psi_b$ and $\psi_a \prec_H \psi' \implies \psi_a \prec_{S_{i+1}} \psi'$.

Assume $\psi' \prec_H \psi_b$ for some $\psi' \in S_i$; then $\psi' \in H_1$. By the choice of ψ_b and ψ_a, all the committed and aborted actions in H_1 are in S_i, and by the choice of ψ, all such actions are in $S'\psi$. Hence, if ψ' is a committed or an aborted action, then $\psi' \in S'\psi$ and, hence, $\psi' \prec_{S_{i+1}} \psi_b$. If ψ' is by thread t, then it is either a committed or an aborted action (and, hence, $\psi' \prec_{S_{i+1}} \psi_b$) or it precedes such an action $\psi'' \in S_i$ by t in H_1: $\psi' \prec_{H_1} \psi''$. Then $\psi' \prec_{S_{i+1}} \psi''$ and $\psi'' \prec_{S_{i+1}} \psi_b$, which implies $\psi' \prec_{S_{i+1}} \psi_b$.

Now assume $\psi_a \prec_H \psi'$ for some $\psi' \in S_i$; then $\psi' \in H_3$. If ψ' is a txbegin action, then $\psi \prec_H \psi'$. Hence, $\psi \prec_{S_i} \psi'$, i.e., $\psi' \in S''$, which implies $\psi_a \prec_{S_{i+1}} \psi'$. If ψ' is by thread t, then it is either a txbegin action (and, hence, $\psi_a \prec_{S_{i+1}} \psi'$) or it follows such an action $\psi'' \in S_i$ by thread t in H_3: $\psi'' \prec_{H_3} \psi'$. Then $\psi'' \prec_{S_{i+1}} \psi'$ and $\psi_a \prec_{S_{i+1}} \psi''$, which implies $\psi_a \prec_{S_{i+1}} \psi'$.

□

D.3 Proof of Lemma 4

Let $H = \mathsf{history}(\tau)$. By assumption, $\mathcal{T}_C \sqsubseteq_{\mathsf{tms}} \mathcal{T}_A$. Hence, there exist histories $H^c \in \mathsf{comp}(H|_{\neg\mathsf{live}})$ and $S \in \mathcal{T}_A$ such that $H^c|_{\mathsf{com}} \sqsubseteq_{\mathsf{op}} S$. Then $H^c = (H|_{\neg\mathsf{live}})H'$ for some H'. Let τ^c be the trace obtained from τ in the same way as H^c is obtained from H: $\tau^c = \tau_0 H'$, where τ_0 is obtained from τ by discarding all live transactions; thus, $\mathsf{history}(\tau^c) = H^c$. It is easy to see that $\tau^c \in [\![P]\!](s)$. Besides, $\tau^c|_{\neg\mathsf{trans}} = \tau|_{\neg\mathsf{trans}}$ and $(\tau|_t)|_{\mathsf{obs}}$ is a prefix of $(\tau^c|_t)|_{\mathsf{obs}}$ for any t. Let $\tau^{na} = \tau^c|_{\neg\mathsf{abortact}}$. By Proposition 1 we get $\tau^{na} \in [\![P]\!](s)$. Since $(H^c|_{\neg\mathsf{abortact}})|_{\neg\mathsf{abortedtx}} = H^c|_{\mathsf{com}} \sqsubseteq_{\mathsf{op}} S$, by Lemma 3, for some history S' we have $\mathsf{history}(\tau^{na}) = H^c|_{\neg\mathsf{abortact}} \sqsubseteq_{\mathsf{op}} S'$ and $S' \in \mathsf{addab}(S)$. Since $S \in \mathcal{T}_A$ and $\mathcal{T}_A$ is closed under immediate aborts (CLP1), we have $S' \in \mathcal{T}_A$. We have $\tau^{na} \in [\![P]\!](s)$; hence, by Lemma 2, there exists a trace $\tau'' \in [\![P]\!](s)$ such that $\mathsf{history}(\tau'') = S' \in \mathcal{T}_A$,

$$\tau''|_{\neg\mathsf{trans}} = \tau^{na}|_{\neg\mathsf{trans}} = \tau^c|_{\neg\mathsf{trans}} = \tau|_{\neg\mathsf{trans}},$$

and $\tau''|_t = \tau^{na}|_t$ for any t. Let τ' be the history obtained from τ'' by discarding the actions in H^c, which are last actions by the corresponding threads. Then

$$\tau'|_{\neg\mathsf{trans}} = \tau''|_{\neg\mathsf{trans}} = \tau|_{\neg\mathsf{trans}},$$

$\tau' \in [\![P]\!](s)$ and, since $\mathcal{T}_A$ is closed under removing transaction responses (CLP2), $\mathsf{history}(\tau') \in \mathcal{T}_A$. Given $\tau''|_t = \tau^{na}|_t$, it is also easy to check that $(\tau'|_t)|_{\mathsf{obs}} = (\tau|_t)|_{\mathsf{obs}}$, as required. □

D.4 Proof of Theorem 1(ii) (Necessity)

Let $\tau\lfloor_i$ denote the prefix of a trace τ containing i actions.

Definition 9. *Two traces τ and τ' are **equivalent up to action identifiers**, denoted $\tau \equiv \tau'$, if $|\tau| = |\tau'|$ and for every $i = 1..|\tau|$, actions $\tau(i)$ and $\tau'(i)$ may differ only in their action identifiers.*

Theorem 1(ii) follows from Lemmas 5 and 6 stated and proved below.

Lemma 5. *Let $\mathcal{T}_C$ and $\mathcal{T}_A$ be TMs such that $\mathcal{T}_C \preceq \mathcal{T}_A$, and $\mathcal{T}_A$ satisfies CLP3 and CLP4. Then*

$$\forall H \in \mathcal{T}_C.\, \exists H^c \in \mathsf{comp}(H|_{\neg\mathsf{live}}), S \in \mathcal{T}_A.\, H^c|_{\mathsf{com}} \sqsubseteq_{\mathsf{op}} S.$$

Proof. Let us choose an integer value $u \neq 1$ which does not appear in H. We use the following shorthands:

- We let m be the largest thread identifier occurring in H.
- We denote by k^t the number of transactions started by thread t in H, i.e., the number of $(\_, t, \mathsf{txbegin})$ actions in H.
- We partition $H|_t$ into k^t subsequences: $H|_t = H_1^t \dots H_{k^t}^t$, where H_i^t is comprised of the actions in the i-th transaction of t. Specifically, $H_i^t(1) = (\_, t, \mathsf{txbegin})$.
- We let c_i^t be the outcome of the i-th transaction of thread t, i.e., $c_i^t = \mathsf{committed}$ or $c_i^t = \mathsf{aborted}$. If the transaction is not completed, c_i^t is undefined.
- We denote by q_i^t the number of call actions of thread t in its i-th transaction, i.e., in H_i^t.
- We let $(\_, t, \mathsf{call}\ o_{i,j}^t.f_{i,j}^t(n_{i,j}^t))$ be the j-th call action of thread t in its i-th transaction.
- We let $(\_, t, \mathsf{ret}(r_{i,j}^t)\ o_{i,j}^t.f_{i,j}^t)$ be the j-th ret action of thread t in its i-th transaction. If the response to $(\_, t, \mathsf{call}\ o_{i,j}^t.f_{i,j}^t(n_{i,j}^t))$ is an $\mathsf{aborted}$ action, we let $r_{i,j}^t = \mathsf{aborted}$. If there is no response to $(\_, t, \mathsf{call}\ o_{i,j}^t.f_{i,j}^t(n_{i,j}^t))$, i.e., the transaction is live and $(\_, t, \mathsf{call}\ o_{i,j}^t.f_{i,j}^t(n_{i,j}^t))$ is its last action, then we let $r_{i,j}^t = u$.
- We denote by $\mathsf{lasttx}(t, i, t')$ the number of transactions of thread t' in H that either committed or aborted before the i-th transaction of thread t started, i.e., the number of $(\_, t', \mathsf{committed})$ and $(\_, t', \mathsf{aborted})$ actions preceding the i-th $(\_, t, \mathsf{txbegin})$ action in H.

For every thread $t = 1..m$ we construct a straight-line command

$$C_H^t = GP_1^t; CP_1^t; GP_2^t; CP_2^t; \dots; GP_{k^t}^t; CP_{k^t}^t, \tag{4}$$

where GP_i^t and CP_i^t are defined in Figure 5. Here the g_i^t variables are global and all others are local. The g_i^t variables are used to monitor the real-time order: g_i^t is written only by thread t and is used to signal that the i-th transaction of thread t ended. The variable $z_{i,t'}^t$ is used to record whether the $\mathsf{lasttx}(t, i, t')$-th transaction of thread t' signaled that it had ended before the i-th transaction of thread t started. As $\mathsf{lasttx}(t, i, t')$ might be 0, we add a dummy variable g_0^t for every thread t. Later in the proof we execute the program from a state in which g_0^t is initialized to 1. The variable w_i^t records whether the i-th transaction of thread t committed or aborted. The variables $y_{i,j}^t$ record the return value of the j-th object method invocation in the i-th transaction of thread t.

Thus, the command $GP_i^t; CP_i^t$ begins by reading the signals of the last transaction of every thread that, according to H, should end before the i-th transaction of thread t starts. It then performs an `atomic` block in which it invokes the sequence of object method invocations induced by H_i^t. After the `atomic` block ends, the command signals the end of the transaction. We define the program P_H as follows:

$$P_H \;=\; C_H^1 \parallel \dots \parallel C_H^m. \tag{5}$$

We now construct a particular trace τ of P_H. We build τ by first constructing a trace τ^t for every sequential command C_H^t and then interleaving the traces $\tau^1, \dots, \tau^m$ in a

$$GP_i^t \quad = \quad z_{i,1}^t := g^1_{\mathsf{lasttx}(t,i,1)}; \texttt{if}(z_{i,1}^t \neq 1)\ \texttt{then fault};$$
$$\ldots$$
$$z_{i,m}^t := g^m_{\mathsf{lasttx}(t,i,m)}; \texttt{if}(z_{i,m}^t \neq 1)\ \texttt{then fault}$$

– If H_i^t is a committed transaction or aborted and visible transaction, then

$$\begin{aligned} CP_i^t \quad = \quad & w_i^t := \texttt{atomic}\ \{\ y_{i,1}^t := o_{i,1}^t.f_{i,1}^t(n_{i,1}^t);\ \texttt{if}(y_{i,1}^t \neq r_{i,1}^t)\ \texttt{then fault}; \\ & \ldots; \\ & y_{i,q_i^t}^t := o_{i,q_i^t}^t.f_{i,q_i^t}^t(n_{i,q_i^t}^t);\ \texttt{if}(y_{i,q_i^t}^t \neq r_{i,q_i^t}^t)\ \texttt{then fault}\ \} \\ & \texttt{if}(w_i^t \neq c_i^t)\ \texttt{then fault else}\ g_i^t := 1 \end{aligned}$$

– If H_i^t is a live transaction or aborted and not visible transaction, then

$$\begin{aligned} CP_i^t \quad = \quad & w_i^t := \texttt{atomic}\ \{\ y_{i,1}^t := o_{i,1}^t.f_{i,1}^t(n_{i,1}^t); \\ & \ldots; \\ & y_{i,q_i^t}^t := o_{i,q_i^t}^t.f_{i,q_i^t}^t(n_{i,q_i^t}^t); \\ & \texttt{fault}\ \} \end{aligned}$$

– If H_i^t is a commit-pending transaction, then

$$\begin{aligned} CP_i^t \quad = \quad & w_i^t := \texttt{atomic}\ \{\ y_{i,1}^t := o_{i,1}^t.f_{i,1}^t(n_{i,1}^t);\ \texttt{if}(y_{i,1}^t \neq r_{i,1}^t)\ \texttt{then fault}; \\ & \ldots; \\ & y_{i,q_i^t}^t := o_{i,q_i^t}^t.f_{i,q_i^t}^t(n_{i,q_i^t}^t);\ \texttt{if}(y_{i,q_i^t}^t \neq r_{i,q_i^t}^t)\ \texttt{then fault}\ \} \end{aligned}$$

Fig. 5. The construction of CP_i^t and GP_i^t for Lemma 5 and Lemma 6 (the case of $t \neq t_0$). For conciseness, we use an extention of the programming language with conditionals without an "else" clause.

particular way. Consider the set of traces $A(C_H^t)t$ of the sequential command C_H^t and let $\tau^t \in A(C_H^t)t$ be the maximal trace without any `fault` actions such that $H|_t = \mathsf{history}(\tau^t)$. The trace τ^t exists, since by construction of C_H^t and the definition of the trace set of a sequential command (Figure 4), there is a trace in $A(C_H^t)t$ for every possible parameter and return value of object method invocations and `atomic` blocks in C_H^t; in particular, $A(C_H^t)t$ contains a trace where the parameters and return values of object method invocations and the return values of transactions are as in $H|_t$. We now partition every τ^t into $\big|H|_t\big|$ subsequences that we later interleave to create τ:

$$\tau^t = \tau_1^t \ldots \tau_{|H|_t|}^t. \tag{6}$$

Formally, for every $i = 1..\big|H|_t\big|$ there is exactly one TM interface action ψ_i^t in τ_i^t and the conditions in Figure 6 hold. This defines τ_i^t uniquely and ensures that, if τ_i^t ends with $\psi_i^t = (\_, t, \mathsf{txbegin})$, then it contains all the actions that are used to read the global signaling variables that precede ψ_i^t in τ^t. The desired trace is constructed by interleaving the subsequences of the traces $\tau^1, \ldots, \tau^m$ according to the order induced by H. Formally,

$$\tau = \tau_{j_1}^{t_1} \ldots \tau_{j_{|H|}}^{t_{|H|}}, \tag{7}$$

$$\tau_i^t = \begin{cases} \_\,\psi_i^t, & \psi_i^t \in \{(\_, t, \mathsf{txbegin}), (\_, t, \mathsf{call}\ o.f(n)), (\_, t, \mathsf{txcommit})\}; \\ \psi_i^t, & \psi_i^t = (\_, t, \mathsf{OK}); \\ \psi_i^t\,(\_, t, y_{i,\_}^t := n), & \psi_i^t = (\_, t, \mathsf{ret}(n)\ o.f) \wedge \mathsf{txof}(\psi_i^t, \tau^t) \in \mathsf{live}(\tau^t); \\ \psi_i^t\,(\_, t, y_{i,\_}^t := n)\,(\_, t, \mathsf{assume}(y_{i,\_}^t = r_{i,\_}^t)), & \\ & \psi_i^t = (\_, t, \mathsf{ret}(n)\ o.f) \wedge \mathsf{txof}(\psi_i^t, \tau^t) \notin \mathsf{live}(\tau^t); \\ \psi_i^t\,(\_, t, w_i := c_i^t)\,(\_, t, \mathsf{assume}(w_i = c_i^t))\,(\_, t, g_i^t := 1), & \\ & \psi_i^t \in \{(\_, t, \mathsf{committed}), (\_, t, \mathsf{aborted})\}. \end{cases}$$

Fig. 6. The construction of τ_i^t for Lemma 5 and Lemma 6 (the case of $t \neq t_0$).

where $H(i) = (\_, t_i, \_)$ and $j_i = \left|(H{\downarrow_i})|_{t_i}\right|$. Note that by construction $\mathsf{history}(\tau) = H$.

Since $\tau^t \in A(C_H^t)t$, we have $\tau \in A(P_H)$. Let s be the state where all the local variables are set to u and for all t, $g_i^t = 0$ for $i \neq 0$ and $g_0^t = 1$. By the construction of τ, we have $\mathsf{eval}(s, \tau) \neq \emptyset$. Then, since $\mathsf{history}(\tau) = H \in \mathcal{T}_C$, we have $\tau \in [\![P_H]\!](s, \mathcal{T}_C)$. Since $\tau \in [\![P_H]\!](s, \mathcal{T}_C)$ and $\mathcal{T}_C \preceq \mathcal{T}_A$, by Definition 7 there exists a trace $\tau' \in [\![P_H]\!](s, \mathcal{T}_A)$ such that $\tau'|_{\neg\mathsf{trans}} = \tau|_{\neg\mathsf{trans}}$ and $S_1 = \mathsf{history}(\tau') \in \mathcal{T}_A$.

Consider a thread t and let T and T' be the i-th transactions in $\tau|_t$ and $\tau'|_t$, respectively. These transactions arise from executing the same commands, and T' might not exist if the commands did not execute in τ'. We now analyze the relationship between T and T'. The construction in Figure 5 ensures the following:

- If T is completed, then so is T', since T is followed in τ by a non-transactional action assigning to g_i^t and $\tau'|_{\neg\mathsf{trans}} = \tau|_{\neg\mathsf{trans}}$. In addition, a completed T is committed if and only if so is T', and in this case the checks done inside the corresponding atomic block ensure that the return values for transactional actions inside T and T' match.
- If T is live, then the `fault` command before the end of the atomic block ensures that T' is live, aborted or does not exist.
- If T is commit-pending, then T' may have any status or may not exist at all. However, if T is visible, then checks inside the atomic block ensure that the return values for transactional actions inside T and T' match.

Let $H_1 = (H|_{\neg\mathsf{live}})|_{\neg\mathsf{abortedtx}}$ and $S_2 = (S_1|_{\neg\mathsf{live}})|_{\neg\mathsf{abortedtx}}$. Since $S_1 \in \mathcal{T}_A$ and $\mathcal{T}_A$ is closed under removing live and aborted transactions (CLP3), $S_2 \in \mathcal{T}_A$. Let p_t', respectively, p_t'' be the index of last $\mathsf{txcommit}$ or $\mathsf{committed}$ action in $H_1|_t$, respectively, $S_2|_t$; if there is no such action, the corresponding index is 0. Let $p_t = \min(p_t', p_t'')$. From the above analysis it follows that $(H_1|_t){\downarrow_{p_t}} \equiv (S_2|_t){\downarrow_{p_t}}$ for any t. Let S_3 be a history obtained from S_2 by renaming the action identifiers such that $S_3 \equiv S_2$, $(H_1|_t){\downarrow_{p_t}} = (S_3|_t){\downarrow_{p_t}}$ and all actions in S_3 that do not belong to $(S_2|_t){\downarrow_{p_t}}$ for some t have identifiers that do not appear in H. Since $S_2 \in \mathcal{T}_A$ and $\mathcal{T}_A$ is closed under renaming action identifiers, we get $S_3 \in \mathcal{T}_A$.

Since $S_3 \in \mathcal{T}_A$ and $\mathcal{T}_A$ is closed under completing commit-pending transactions (CLP4), we get that there exists a history $S_4 \in \mathsf{nicomp}(S_3) \cap \mathcal{T}_A$. Let S_3' be subsequence of S_4 that contains only $\mathsf{committed}$ and $\mathsf{aborted}$ actions which do no appear in S_3; without loss of generality we can assume that identifiers of actions in S_3' do not appear in H. Let S_3'' be the subsequence of $\mathsf{committed}$ actions in S_3 that are not in

H_1. The history $(H|_{\neg\mathsf{live}})S_3''S_3'$ contains only visible transactions. We construct the desired history H^c by aborting all the commit-pending transactions in $(H|_{\neg\mathsf{live}})S_3''S_3'$. Let $H^c = (H|_{\neg\mathsf{live}})S_3''S_3'H_a$, where H_a consists of actions $(\_, t, \mathsf{aborted})$ for every thread t ending with a commit-pending transaction in $(H|_{\neg\mathsf{live}})S_3''S_3'$; these actions have unique identifiers that do not appear in H. We have that $H^c \in \mathsf{comp}(H|_{\neg\mathsf{live}})$, for the following reasons:

(i) S_3'' completes the commit-pending transactions in H that get committed in S_1.
(ii) S_3' completes the commit-pending transactions in H that stay commit-pending in S_1.
(iii) H_a aborts the commit-pending transactions in H that become live or aborted in S_1 or do not appear there at all.

Let $S = S_4|_{\mathsf{com}}$. Since $S_4 \in \mathcal{T}_A$ has only completed transactions and $\mathcal{T}_A$ is closed under removing live and aborted transactions (CLP3), we get that $S = S_4|_{\mathsf{com}} \in \mathcal{T}_A$. We now show $H^c|_{\mathsf{com}} \sqsubseteq_{\mathsf{op}} S$, thus completing the proof. We first show that $\forall t.\, (H^c|_t)|_{\mathsf{com}} = S|_t$ by considering several cases.

– $H|_t$ does not contain commit-pending transactions. Then neither does $S|_t$, $H_1|_t = S_3|_t$, and $S_3' = S_3'' = H_a = \varepsilon$. Hence,

$$(H^c|_t)|_{\mathsf{com}} = (H_1|_t)|_{\mathsf{com}} = (S_3|_t)|_{\mathsf{com}} = (S_4|_t)|_{\mathsf{com}} = S|_t.$$

– $H|_t$ ends with a commit-pending transaction and $(S_3''S_3'H_a)|_t = S_3''|_t = (\_, t, \mathsf{committed})$. Then $H_1|_t = (H_1|_t)\downarrow_{p_t}$ and $S_3|_t = ((S_3|_t)\downarrow_{p_t})(S_3''|_t)$. Hence,

$$\begin{aligned}(H^c|_t)|_{\mathsf{com}} = ((H_1|_t)\,(S_3''|_t))|_{\mathsf{com}} = (((H_1|_t)\downarrow_{p_t})\,(S_3''|_t))|_{\mathsf{com}} = \\ (((S_3|_t)\downarrow_{p_t})\,(S_3''|_t))|_{\mathsf{com}} = (S_3|_t)|_{\mathsf{com}} = (S_4|_t)|_{\mathsf{com}} = S|_t.\end{aligned}$$

– $H|_t$ ends with a commit-pending transaction and $(S_3''S_3'H_a)|_t = S_3'|_t$. Then $H_1|_t = S_3|_t$. Hence,

$$(H^c|_t)|_{\mathsf{com}} = ((H_1|_t)\,(S_3'|_t))|_{\mathsf{com}} = ((S_3|_t)\,(S_3'|_t))|_{\mathsf{com}} = (S_4|_t)|_{\mathsf{com}} = S|_t.$$

– $H|_t$ ends with a commit-pending transaction and $(S_3''S_3'H_a)|_t = H_a|_t = (\_, t, \mathsf{aborted})$. Then $((H_1|_t)\,(H_a|_t))|_{\mathsf{com}} = ((H_1|_t)\downarrow_{p_t})|_{\mathsf{com}}$ and $S_3|_t = (S_3|_t)\downarrow_{p_t}$. Hence,

$$\begin{aligned}(H^c|_t)|_{\mathsf{com}} = ((H_1|_t)\,(H_a|_t))|_{\mathsf{com}} = ((H_1|_t)\downarrow_{p_t})|_{\mathsf{com}} = \\ ((S_3|_t)\downarrow_{p_t})|_{\mathsf{com}} = (S_3|_t)|_{\mathsf{com}} = (S_4|_t)|_{\mathsf{com}} = S|_t.\end{aligned}$$

This shows $\forall t.\, (H^c|_t)|_{\mathsf{com}} = S|_t$. Finally, for every completed transaction T in τ, the value of g_i^t is set to 1 after the transaction completes and is read before every transaction T' that begins after the completion of T. Hence, the transaction in τ' corresponding to T completes before the transaction corresponding to T' begins. Then the real-time order in H^c is preserved in S, which implies $H^c|_{\mathsf{com}} \sqsubseteq_{\mathsf{op}} S$. □

Lemma 6. *Let $\mathcal{T}_C$ and $\mathcal{T}_A$ be TMs such that $\mathcal{T}_C \preceq \mathcal{T}_A$ and $\mathcal{T}_A$ satisfies CLP3 and CLP4. Let $H = H'\psi \in \mathcal{T}_C$, where ψ is a response action that is not a* committed *or* aborted *action. There exist $H^c \in \mathsf{cTMSpast}(H)$ and $S \in \mathcal{T}_A$ such that $H^c \sqsubseteq_{\mathsf{op}} S$.*

$$
\begin{array}{rcl}
GP_i^{t_0} & = & z_{i,1}^{t_0} := g^1_{\mathsf{lasttx}(t_0,i,1)}; \texttt{if}(z_{i,1}^{t_0} \neq 1) \texttt{ then } \mathit{mismatch} := 1; \\
& & \ldots \\
& & z_{i,m}^{t_0} := g^m_{\mathsf{lasttx}(t_0,i,m)}; \texttt{if}(z_{i,m}^{t_0} \neq 1) \texttt{ then } \mathit{mismatch} := 1
\end{array}
$$

– For $i \neq k^{t_0}$, $CP_i^{t_0}$ is constructed as follows:

$$
\begin{array}{rcl}
CP_i^{t_0} = & w_i^{t_0} := \texttt{atomic}\,\{\, y_{i,1}^{t_0} := o_{i,1}^{t_0}.f_{i,1}^{t_0}(n_{i,1}^{t_0});\ \texttt{if}(y_{i,1}^{t_0} \neq r_{i,1}^{t_0}) \texttt{ then } \mathit{mismatch} := 1; \\
& \qquad \ldots; \\
& \qquad y_{i,q_i^{t_0}}^{t_0} := o_{i,q_i^{t_0}}^{t_0}.\, f_{i,q_i^{t_0}}^{t_0}(n_{i,q_i^{t_0}}^{t_0});\ \texttt{if}(y_{i,q_i^{t_0}}^{t_0} \neq r_{i,q_i^{t_0}}^{t_0}) \texttt{ then } \mathit{mismatch} := 1\,\} \\
& \texttt{if}(w_i^{t_0} \neq c_i^{t_0}) \texttt{ then } \mathit{mismatch} := 1 \texttt{ else } g_i^{t_0} := 1
\end{array}
$$

– $CP_{k^{t_0}}^{t_0}$ is constructed as follows:

$$
\begin{array}{rcl}
CP_{k^{t_0}}^{t_0} & = & w_{k^{t_0}}^{t_0} := \texttt{atomic}\,\{\, y_{k^{t_0},1}^{t_0} := o_{k^{t_0},1}^{t_0}.f_{k^{t_0},1}^{t_0}(n_{k^{t_0},1}^{t_0}); \\
& & \qquad \texttt{if}(y_{k^{t_0},1}^{t} \neq r_{k^{t_0},1}^{t}) \texttt{ then } \mathit{mismatch} := 1; \\
& & \qquad \ldots; \\
& & \qquad y_{k^{t_0},q_{k^{t_0}}^{t_0}}^{t_0} := o_{k^{t_0},q_{k^{t_0}}^{t_0}}^{t_0} \cdot f_{k^{t_0},q_{k^{t_0}}^{t_0}}^{t_0}\,(n_{k^{t_0},q_{k^{t_0}}^{t_0}}^{t_0}); \\
& & \qquad \texttt{if}(y_{k^{t_0},q_{k^{t_0}}^{t}}^{t} \neq r_{k^{t_0},q_{k^{t_0}}^{t}}^{t}) \texttt{ then } \mathit{mismatch} := 1; \\
& & \qquad \texttt{if}(\mathit{mismatch} \neq 1) \texttt{ then } \texttt{fault}\,\}
\end{array}
$$

Fig. 7. The construction of $CP_i^{t_0}$ and $GP_i^{t_0}$ for Lemma 6. For conciseness, we use an extention of the programming language with conditionals without an "else" clause.

Proof. We reuse the notation introduced in the proof of Lemma 5. As in that case, we construct a program P_H as defined by (5) and (4). However, the commands GP_i^t and CP_i^t are constructed somewhat differently. Let $\psi = (\_, t_0, \_)$. For $t \neq t_0$ we again define CP_i^t and GP_i^t as shown in Figure 5; however, we define $CP_i^{t_0}$ and $GP_i^{t_0}$ as shown in Figure 7. Thus, instead of faulting immediately upon detecting a mismatch with the history $H|_{t_0}$, thread t_0 in P_H sets the local variable $\mathit{mismatch}$ to 1; at the end of its execution, the thread checks the variable and faults if there has *not* been a mismatch.

Similarly to the proof of Lemma 5, we construct a trace τ of P_H. For $t = 1..m$ let $\tau^t \in A(C_H^t)t$ be the maximal trace in $A(C_H^t)t$ such that $H|_t = \mathsf{history}(\tau_c^t)$; then τ^{t_0} ends with a `fault`. We partition every τ^t into subsequences τ_i^t as defined by (6), but with the conditions in Figure 6 changed so that the case of $t = t_0$ and $\psi_i^{t_0} = \psi$ is treated specially: in this case

$$\tau_i^t = \psi\,(\_, t, y_{i,\_}^t := n)\,(\_, t, \mathsf{assume}(y_{i,\_}^t = r_{i,\_}^t))\,(\_, t, \mathsf{assume}(\mathit{mismatch} \neq 1))\,(\_, t, \texttt{fault}).$$

Then we let τ be defined by (7), so that $\mathsf{history}(\tau) = H$. Let s be the initial state chosen as in Lemma 5. As before, we have $\tau \in [\![P_H]\!](s, \mathcal{T}_C)$. Since τ ends with a `fault` by t_0 and $\mathcal{T}_C \preceq \mathcal{T}_A$, by Definition 7 there exists $\tau' \in [\![P_H]\!](s, \mathcal{T}_A)$ that also ends with a `fault` by t_0. Then $S_1 = \mathsf{history}(\tau') \in \mathcal{T}_A$.

Since both τ and τ' end with a `fault` by t_0, the checks inside this thread ensure that

$$(H|_{\neg\mathsf{abortedtx}})|_{t_0} \equiv (S_1|_{\neg\mathsf{abortedtx}})|_{t_0}. \tag{8}$$

Consider threads $t \neq t_0$ and t'. Let T and T_1 be the i-th transactions in $H|_t$ and $S_1|_t$, respectively and let T' and T'_1 be j-th transactions in $H|_{t'}$ and $S_1|_{t'}$ (T_1 and T'_1 may not exist). The construction of P_H ensures the following:

1. If T_1 is visible, then so is T. Indeed, in this case T cannot be live or aborted and not visible because of the `fault` command before the end of the corresponding atomic block (Figure 5). Furthermore, because of the checks done inside the atomic block, in this case the return values for transactional actions inside T and T' match.
2. If $T' \prec_H T$ and T_1 exists, then so does T'_1, $T'_1 \prec_{S_1} T_1$, and T' is committed if and only if so is T'_1. This is because before the transaction corresponding to T_1 starts in τ' there is a check that a $g_k^{t'}$ variable is 1, and this variable is assigned to 1 only if the checks for return values inside the transaction corresponding to T'_1 and for the status of this transaction have passed.

Let $S_2 = (S_1|_{\psi\cup\neg\mathsf{live}})|_{\neg\mathsf{abortedtx}}$, where $\cdot|_{\psi\cup\neg\mathsf{live}}$ is the projection to actions by non-live transactions that nevertheless includes the transaction of ψ. Since $S_1 \in \mathcal{T}_A$ and $\mathcal{T}_A$ is closed under removing live and aborted transactions (CLP3), $S_2 \in \mathcal{T}_A$. Since $S_2 \in \mathcal{T}_A$ and $\mathcal{T}_A$ is closed under completing commit-pending transactions (CLP4), we get that there exists a history $S_3 \in \mathsf{nicomp}(S_2) \cap \mathcal{T}_A$. Let $S_4 = S_3|_{\psi\cup\mathsf{com}}$, where $\cdot|_{\psi\cup\mathsf{com}}$ projects to committed transactions and the transaction of ψ. Since $S_3 \in \mathcal{T}_A$ does not have commit-pending transactions and $\mathcal{T}_A$ is closed under removing live and aborted transactions (CLP3), we get that $S_4 \in \mathcal{T}_A$.

Let H_1 be the subsequence of H consisting of those transactions for which the matching transactions in S_1 are included into S_4. Then from item 1 above and (8) we get that

$$\mathsf{tx}(H_1) \subseteq \{\mathsf{txof}(\psi, H)\} \cup \mathsf{visible}(H).$$

Consider $T, T' \in \mathsf{tx}(H_1)$ such that $T' \prec_{H_1} T$. Then there exist matching transactions $T_1, T'_1 \in \mathsf{tx}(S_1)$. It is easy to see that T'_1 has to be committed; then by item 2 above we get that T' is committed as well. Conversely, consider $T \in \mathsf{tx}(H_1)$ and a committed $T' \in \mathsf{tx}(H)$ such that $T' \prec_H T$. Since $T \in \mathsf{tx}(H_1)$, there exists a matching transaction $T_1 \in \mathsf{tx}(S_1)$. Then by item 2 above there also exists a matching transaction $T'_1 \in \mathsf{tx}(S_1)$ for T' and T'_1 is committed. Hence, it is included into S_4 and, thus, $T' \in \mathsf{tx}(H_1)$. We have just shown that $H_1 \in \mathsf{TMSpast}(H)$.

Let $H^c \in \mathsf{ccomp}(\mathsf{com}(H'_1))$. Then $H^c \in \mathsf{cTMSpast}(H)$. From (8) we get $(H_1|_{t_0}) \equiv (S_4|_{t_0})$. For $t \neq t_0$ let p_t be the index of last `txcommit` action in $S_4|_t$. Then from items 1 and 2 above it follows that $(H_1|_t)\downarrow_{p_t} \equiv (S_4|_t)\downarrow_{p_t}$ for any t. Since S_4 contains only committed transactions, this implies $H_1|_t \equiv S_4|_t$ for any t. Let S be the history obtained from S_4 by renaming action identifiers such that for any t we have $H^c|_t = S|_t$. Since $S_4 \in \mathcal{T}_A$ and $\mathcal{T}_A$ is closed under renaming action identifiers, we get $S \in \mathcal{T}_A$. By item 2 above, the real-time order in H^c is preserved between the corresponding transactions in S_1. This gives us $H^c|_{\mathsf{com}} \sqsubseteq_{\mathsf{op}} S$, as required. □