Characterizing Transactional Memory Consistency Conditions Using Observational Refinement

HAGIT ATTIYA, Technion—Israel Institute of Technology
ALEXEY GOTSMAN, IMDEA Software Institute
SANDEEP HANS, Technion—Israel Institute of Technology
NOAM RINETZKY, Tel Aviv University

Transactional memory (TM) facilitates the development of concurrent applications by letting a programmer designate certain code blocks as atomic. The common approach to stating TM correctness is through a consistency condition that restricts the possible TM executions. Unfortunately, existing consistency conditions fall short of formalizing the intuitive semantics of atomic blocks through which programmers use a TM. To close this gap, we formalize programmer expectations as observational refinement between TM implementations. This states 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 consistency condition, is equivalent to observational refinement for a programming language where local variables are rolled back upon a transaction abort. We thereby establish that TMS is the weakest acceptable condition for this case. We then propose a new consistency condition, called *Strong Transactional Memory Specification (STMS)*, and show that it is equivalent to observational refinement for a language where local variables are not rolled back upon aborts. Finally, we show that under certain natural assumptions on TM implementations, STMS is equivalent to a variant of a well-known condition of opacity.

Our results suggest a new approach to evaluating TM consistency conditions and enable TM implementors and language designers to make better-informed decisions.

CCS Concepts: • **Theory of computation → Parallel computing models**; **Program specifications**; *Program verification*; Operational semantics;

Additional Key Words and Phrases: Transactions, correctness conditions, opacity, Transactional Memory Specification, TMS, Strong Transactional Memory Specification, STMS, atomicity

S. Hans is currently with IBM Research, India.

This article combines and extends results that appeared in preliminary form in Proceedings of the 32nd Annual ACM Symposium on Principles of Distributed Computing (PODC), ACM, New York, 2013, pp. 309–318 and in Proceedings of the 28th International Symposium on Distributed Computing (DISC), Springer-Verlag, Berlin, Germany, 2014, pp. 376–390.
This work was partially supported by EU FP7 projects TRANSFORM (238639) and ADVENT (308830), and by the Broadcom Foundation and Tel Aviv University Authentication Initiative.

Authors' addresses: H. Attiya, Department of Computer Science, Technion, Haifa 32000, Israel; email: hagit@cs.technion.ac.il; A. Gostman, IMDEA Software Institute, Campus Montegancedo s/n, 28223-Pozuelo de Alarcon, Madrid, Spain; email: Alexey.Gotsman@imdea.org; N. Rinetzky, School of Computer Science, Tel Aviv University, Tel Aviv 69978, Israel; email: maon@cs.tau.ac.il; S. Hans, IBM India Research Laboratory, ISID Campus, Plot No. 4, Block C, Institutional Area, Vasant Kunj Phase-II, New Delhi-110070, India; email: shans001@in.ibm.com.

https://doi.org/10.1145/3131360

ACM Reference format:
Hagit Attiya, Alexey Gotsman, Sandeep Hans, and Noam Rinetzky. 2017. Characterizing Transactional Memory Consistency Conditions Using Observational Refinement. *J. ACM* 65, 1, Article 2 (December 2017), 44 pages.
https://doi.org/10.1145/3131360

1 INTRODUCTION

Transactional memory (TM) facilitates the development of concurrent applications by letting the programmer designate certain code blocks as *atomic* (Herlihy and Moss 1993). TM allows developing a program and reasoning about its correctness as if each atomic block executes as a *transaction*—atomically and without interleaving with other blocks—even though in reality the blocks can be executed concurrently. Figure 1 demonstrates the simplicity of implementing a concurrent (bounded array-based) stack using TM. This is done using two *transactional objects*, managed by the TM: a `Top` variable and an array `Items` of size `N`. The transaction of thread 1 pushes a new item onto the stack, and the transaction of thread 2 pops a value off the top of the stack. In this case, we can perform several operations accessing `Top` and `Items` in a single transaction, with the TM guaranteeing their atomicity.

Many TM implementations have been proposed (e.g., Dalessandro et al. (2010), Dice et al. (2006), Harris et al. (2010), Herlihy et al. (2003), Marathe et al. (2005), and Riegel et al. (2006)). They use myriad design approaches that, for efficiency, may execute transactions concurrently, even though they aim to provide the programmer with an illusion that the transactions are executed atomically. This illusion is not always perfect, as transactions may abort, typically due to conflicts with concurrently running ones, and need to be restarted (see the transaction of thread 2 in Figure 1).

How can we be sure that a TM indeed implements atomic blocks correctly? The common approach to stating TM correctness is through a *consistency condition* that restricts the possible TM executions. A TM consistency condition can be similar to a database consistency condition, such as *serializability* (Papadimitriou 1979). The latter requires that the results of concurrently executing a set of committed transactions could be obtained if these transactions executed atomically in some order according to the sequential semantics of transactional objects. However, there is a subtlety: serializability provides no guarantees for *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. However, in the setting of TM, this is often unsatisfactory.

For example, the accesses to the array `Items` in Figure 1 are only safe when $0 \leq \texttt{Top} \leq \texttt{N}$—an invariant that is preserved by the transactions in the figure when they are executed atomically. If the TM allows (e.g., the transaction of thread 2 to read $\texttt{Top} > \texttt{N}$), this will lead to the program *faulting* due to an out-of-bounds array access. Allowing such a behavior is undesirable.

There have been several proposals of TM consistency conditions that constrain the behavior of live transactions. *Opacity* was the first one of them (Guerraoui and Kapalka 2008). Roughly speaking, opacity requires that for any sequence of interactions between the program and the TM, dubbed a *history*, there exists another *justifying* history where

(i) the interactions of every separate thread are the same as in the original history,
(ii) the order of nonoverlapping transactions in the original history is preserved, and
(iii) each transaction executes atomically and yields results consistent with the sequential semantics of transactional objects.

Opacity thus constrains the behavior of live, as well as aborted, transactions by including actions inside them into the justifying history. Opacity has been followed by more proposals of consis-

```
Thread 1: push(v)
pos := −1;
res1 := atomic {
    pos := Top;
    if (pos ≤ N − 1) {
        Items[pos] := v;
        Top := pos + 1
    } else {
        abort } };
g := pos
```

```
Thread 2: pop()
r := 0;
res2 := abort;
while (res2 == abort) {
    res2 := atomic {
        if (1 ≤ Top) {
            Top := Top − 1;
            r := Items[Top]
        } else {
            r := EMPTY } } }
```

Fig. 1. Implementation of a concurrent bounded stack using TM. Note that thread 1 explicitly aborts its transaction if the stack is full.

tency conditions that weakened it (*Transactional Memory Specification* (TMS) (Doherty et al. 2013), *Virtual World Consistency* (VWC) (Imbs and Raynal 2012)) or strengthened it (*DU-opacity* (Attiya et al. 2013), *Transactional Memory Specification 2* (TMS2) (Doherty et al. 2013)).

Given the plethora of proposals, it is unclear which condition one should use. An ideal TM consistency condition should satisfy two desiderata. On one hand, it should be strong enough to satisfy the intuitive expectations of the programmer and, in particular, to disallow undesirable behaviors, such as the preceding out-of-bounds array access. On the other hand, the consistency condition should put minimal restrictions on TM implementations needed to achieve this, so as to allow as many optimizations as possible. The first contribution of this article is to propose a formal framework for systematically evaluating TM consistency conditions according to these criteria. Our key insight is to formalize the intuitive expectations of a programmer using *observational refinement* (He et al. 1986, 1987) between TM implementations. Consider two TM implementations—a *concrete* one, such as an efficient TM, and an *abstract* one, such as a TM executing each atomic block atomically. Informally, the concrete TM *observationally refines* the abstract one for a given programming language if every behavior of any program P in this language that a user can observe when P uses 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 $0 \leq \texttt{Top} \leq \texttt{N}$ in Figure 1) using the expected intuitive semantics formalized by the abstract TM. Observational refinement implies that the conclusions (e.g., the safety of array accesses) will carry over to the case when P uses the concrete TM. Thus, if we formulate a consistency condition as a relation between a concrete and an abstract TM, then the condition ideal for a given programming language should be equivalent to observational refinement for this language.

The notion of observational refinement depends on which aspects of program behavior we consider user observable. In this article, we consider observable the sequence of actions performed outside transactions in finite program computations, and whether the program faults or not (inside or outside a transaction). This choice is motivated by the fact that input-output actions in programs using TMs are done outside transactions, unless the TM supports special transactions that are guaranteed not to abort (Spear et al. 2008; Welc et al. 2008); we do not consider such transactions in this article. Since we consider only finite computations, our notion of observation allows a programmer to reason about safety, but not about liveness properties (see Section 9 for discussion).

The next contribution of this article is to identify the TM consistency conditions equivalent to observational refinement for particular programming languages. As we show, the key consideration is how the language treats local variables modified by a transaction (e.g., pos in Figure 1) if the transaction aborts: whether it rolls back the variables to the values they had when the transaction started or leaves them as they are. There are programming languages that behave in both ways:

Scala TM (Scala STM Expert Group 2012) rolls back local variables, whereas most TM systems do not. We prove the following:

- —When local variables are rolled back upon transaction abort, observational refinement is equivalent to a relational variant of the previously proposed TMS (Doherty et al. 2013), which relates a concrete and an abstract TM.
- —When local variables are not rolled back upon transaction abort, observational refinement is equivalent to a new consistency condition that we propose, called *Strong Transactional Memory Specification (STMS)*.

We show that under certain natural assumptions on the abstract TM, STMS is indeed stronger than our relational variant of TMS, and STMS is weaker than an *opacity relation*, a variant of opacity (Guerraoui and Kapalka 2008) that requires every history of the concrete TM to have a justifying history of the abstract TM satisfying the conditions (i) and (ii) presented earlier. We furthermore show that STMS implies the opacity relation under assumptions on the *concrete* TM that require it to satisfy a liveness property similar to lock-freedom (Herlihy and Shavit 2008) and to allow threads to explicitly abort their transactions (like thread 1 in Figure 1).

To concentrate on the core goal of this article, the programming languages we consider do not allow transaction nesting and assume a static separation of transactional and nontransactional shared memory. Extending our development to lift these restrictions is an interesting avenue for future work.

Even in this simple setting, establishing a connection between the consistency conditions and observational refinement is a challenging task, which is due to the subtle way in which TMS and STMS weaken opacity. 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 and STMS relax this requirement by requiring only a subset of completed transactions in the concrete history to be justified by a single abstract one obeying (i) through (ii) shown earlier. TMS and STMS differ in that the former excludes all aborted transactions from this particular check. To specify the behavior of live transactions, TMS and STMS require each response in a live transaction to be justified by a separate abstract TM history, which may be different for different transactions. The constraints on the choice of this abstract history are subtle. Somewhat counterintuitively, it can include transactions that aborted in the concrete history, with their status changed to committed, and exclude some that committed; however, this is subject to certain carefully chosen constraints. The flexibility in the choice of the abstract history is meant to allow the concrete TM implementation to perform as many optimizations as possible. However, it is not straightforward to establish that this flexibility does not invalidate observational refinement (and hence the informal guarantees that programmers expect from a TM) or that the consistency definitions cannot be weakened further.

Our results ensure that this is indeed the case and motivate the definitions of the consistency conditions. Informally, using a separate history for every live transaction in TMS and STMS does not invalidate observational refinement because the only aspect of the behavior of live transactions that users can observe is whether one of them faults or not. For observational refinement, we need to prove that a fault inside a live 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 single live transaction can be justified by some abstract history; different transactions can be justified by different histories. The link to observational refinement similarly explains the differences between TMS and STMS. If local variables are not rolled back when transactions abort, then threads can communicate to each other the observations they make inside aborted transactions about the state of transactional objects. For example, even when the transaction of thread 1

in Figure 1 aborts, the thread may learn the number of elements in the stack through the local variable pos. This knowledge may then be communicated to other threads through global variables, such as g in Figure 1, and may affect their future behavior. This is the reason for STMS requiring a single justifying history not only for committed transactions (as in TMS) but also aborted ones.

Technically, we prove that TMS and STMS are sufficient for observational refinement for the respective programming languages 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 other transactions that are allowed by the consistency conditions. Proving that TMS and STMS are 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 the consistency conditions 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.

2 PROGRAMMING LANGUAGE SYNTAX

We consider a language with programs consisting of a fixed, but arbitrary, number m of *threads*, identified by $\mathsf{ThreadID} = \{1, \ldots, m\}$. Every *thread* $t \in \mathsf{ThreadID}$ has a private 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 TM, 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 the language is as follows:

$$
\begin{aligned}
C &::= c \mid C; C \mid \texttt{if}\,(b)\,\{C\}\,\texttt{else}\,\{C\} \mid \texttt{while}\,(b)\,\{C\} \mid \\
&\qquad x := \texttt{atomic}\,\{C\} \mid x := o.f(e) \mid \texttt{abort} \\
P &::= C_1 \parallel \ldots \parallel C_m,
\end{aligned}
$$

where b and e denote Boolean and integer expressions over local variables, left unspecified. The code of threads in a program P is given by *sequential commands* $C_1, \ldots, C_m$. These include *primitive commands* c from a set PComm, sequential compositions, conditionals, loops, `atomic` blocks, object method invocations, and a special `abort` command that can only be used inside an `atomic` block and whose effect is to abort the corresponding transaction. Primitive commands are meant to execute atomically. We do not fix their set PComm but assume that it at least includes assignments to local and global variables, a `skip` command that does nothing, 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 or out-of-bounds memory access.

An *atomic block* $x := \texttt{atomic}\,\{C\}$ executes C as a *transaction*, which the TM can *commit* or *abort*. The TM's decision is returned in the local variable x, which gets assigned distinguished values committed or aborted. We forbid nested atomic blocks and, hence, nested transactions. Inside an atomic block (and only there), the program can invoke methods on transactional objects, as in $x := o.f(e)$. Here the expression e gives the value of the method parameter, and x gets assigned the return value after the method terminates. The transactional system may decide to abort a transaction initiated by $x := \texttt{atomic}\,\{C\}$ not only upon reaching the end of the `atomic` block but also during the execution of a method on a transactional object. Once this happens, the execution of C terminates. Transactions can be aborted explicitly using the `abort` command (e.g., as done by

thread 1 in Figure 1 if the array is full). A typical pattern of using the transactional system is to execute a transaction repeatedly until it commits, as done by thread 2 in Figure 1.

We assume that a thread cannot access global variables inside atomic blocks and cannot access local variables of other threads. Thus, we accordingly restrict the expressions used in the conditions of `if` and `while` commands. To restrict accesses by primitive commands, we partition the set PComm − {`fault`} into $2m$ classes: PComm − {`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 ($\mathsf{LVar}_t \uplus$ GVar). We formalize these restrictions in Section 4. We then require a thread t to use only primitive commands from $\mathsf{LPcomm}_t \uplus \mathsf{GPcomm}_t \uplus$ {`fault`} and to use only those from $\mathsf{LPcomm}_t \uplus$ {`fault`} inside atomic blocks.

We note that whereas transactional objects are managed by the transactional system, global variables are not. Thus, threads can communicate via the transactional system inside `atomic` blocks and directly via global variables outside them. In the following, we define two variants of programming language semantics differing in the treatment of local variables upon a transaction abort: in one case, they are rolled back to the values they had when the transaction started, and in the other case, left as they are. Thus, in the latter case, aborted transactions can communicate information to the following nontransactional code. Local variables are never rolled back when a transaction commits.

3 MODEL OF COMPUTATIONS

We model a program computation by a *trace*, which is a finite sequence of *actions*, each describing a single computation step.[1]

Definition 3.1. Let ActionId be a set of action identifiers. A *primitive action* χ has the form (a, t, c), where $a \in$ ActionId, $t \in$ ThreadID, and $c \in$ PComm is a primitive command. A *TM interface action* ψ has one of the following forms:

Request Actions	Corresponding Response Actions
$(a, t, \text{txbegin})$	(a', t, OK) | $(a', t, \text{aborted})$
$(a, t, \text{txcommit})$	$(a', t, \text{committed})$ | $(a', t, \text{aborted})$
$(a, t, \text{txabort})$	$(a', t, \text{aborted})$
$(a, t, \text{call } o.f(n))$	$(a', t, \text{ret}(n')\ o.f)$ | $(a', t, \text{aborted})$

where $a, a' \in$ ActionId, $t \in$ ThreadID, $o \in$ Obj, $f \in$ Method, and $n, n' \in \mathbb{Z}$. We use φ to range over both primitive and TM interface actions.

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. If this successfully starts a transaction, the TM responds with an OK action. A txcommit action is generated when a transaction tries to commit upon exiting an `atomic` block; if this is successful, the TM responds with a committed action. Actions call and ret denote, respectively, a call to and a return from an invocation of a method f on a transactional object o and are annotated with the method parameter n or return value n'. The TM may abort a transaction at any point when it is in control; this is recorded by an aborted response action.

[1] We do not consider infinite computations; see Section 9 for a discussion.

Notation	Description
ε	An empty trace
$\tau(i)$	The i-th element of τ
$\lvert\tau\rvert$	The length of τ
$\tau\downarrow_i$	The prefix of τ containing i actions
$\tau\lvert_t$	The projection of τ onto actions of thread t
$\tau\lvert_{\neg t}$	The projection of τ onto actions of threads other than t
$\tau\lvert_o$	The projection of τ onto call and ret actions on object o
$\tau_1\tau_2$	The concatenation of τ_1 and τ_2
$\varphi \in \tau$	φ ***is in*** τ: $\tau = \_\varphi\_$
$\tau_1\lvert_{\neg\tau_2}$	The subsequence of τ_1 comprised of the actions that are not in τ_2, i.e., $\psi \in \tau_1\lvert_{\neg\tau_2} \iff (\psi \in \tau_1 \wedge \psi \notin \tau_2)$
$\tau \equiv \tau'$	τ and τ' are ***equivalent***: $\lvert\tau\rvert = \lvert\tau'\rvert$ and for every $i = 1..\lvert\tau\rvert$ actions $\tau(i)$ and $\tau'(i)$ may differ only in their action identifiers
history(τ)	The projection of τ to TM interface actions

Fig. 2. Notation for traces.

In addition to interface actions, we have actions of the form (a, t, c), which denote the execution of a primitive command c by thread t. To denote the evaluation of conditions in `if` and `while` statements, we assume that the sets LPcomm_t contain special primitive commands assume(b), where b is a Boolean expression over local variables of thread t, defining the condition. We state their semantics formally in Section 4.2; informally, 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.

We call a trace containing only TM interface actions a *history*. We use τ to range over traces and H, S to range over histories. We denote the set of all traces by Traces and the set of all histories by History. We denote irrelevant expressions by _ and use the notation for traces shown in Figure 2.

Programs in our programming language do not generate arbitrary traces, but only those satisfying certain conditions, summarized in the following definition.

Definition 3.2. A trace τ is *well-formed* if

(i) every action in τ has a unique identifier: if $\tau = \_\,(a_1, \_, \_)\,\_\,(a_2, \_, \_)\,\_$ 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 alternate correctly: for every thread t, history$(\tau)\lvert_t$, consists of alternating request and corresponding response actions, starting with a request action;
(iv) requests must be answered before calling a primitive command: for every thread t, $\tau\lvert_t$ does not contain a request action immediately followed by a primitive action;
(v) actions denoting the beginning and end of transactions alternate correctly: for every thread t, in the projection of $\tau\lvert_t$ to txbegin, committed, and aborted actions, txbegin alternates with committed or aborted, starting with txbegin;
(vi) call, ret, txcommit, and txabort actions occur only inside transactions: for every thread t, if $\tau\lvert_t = \tau_1\psi\tau_2$ for a call, ret, txcommit, or txabort action ψ, then $\tau_1 = \tau_1'\,(\_, t, \text{txbegin})\,\tau_1''$ for some τ_1' and τ_1'' such that τ_1'' does not contain committed or aborted actions;

Notation	Description
$\tau\vert_{\mathsf{trans}}$	The projection of τ to transactional actions
$\tau\vert_{\neg\mathsf{trans}}$	The projection of τ to non-transactional actions
$\tau\vert_{\neg\mathsf{aborted}}$	The projection of τ excluding aborted transactions
$\tau\vert_{\neg\mathsf{live}}$	The projection of τ excluding live transactions
$\tau\vert_{\neg\mathsf{abortact}}$	The trace obtained from τ by removing all actions inside aborted transactions
$\mathsf{txof}(\varphi, \tau)$	The *transaction of* φ *in* τ: the subsequence of τ comprised of all actions that are in the same transaction in τ as φ (undefined if φ is non-transactional)
$T \in \tau$	T *is in* τ: $\tau\vert_t = \tau_1 T \tau_2$ for some t, τ_1 and τ_2, where T is a transaction and either T is completed or τ_2 is empty

Fig. 3. Notation for transactions.

(vii) commands in τ do not access local variables of other threads: if $(\_, t, c) \in \tau$, then $c \in \mathsf{LPcomm}_t \uplus \mathsf{GPcomm}_t \uplus \{\mathtt{fault}\}$;
(viii) commands in τ do not access global variables inside a transaction: for every thread t, if $\tau|_t = \tau_1\,(\_, t, c)\,\tau_2$ 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.

A history is well formed if it is well formed as a trace. We denote the set of all well-formed traces by WfTraces and the set of all well-formed histories by WfHistory.

Definition 3.3. A *transaction* T is a nonempty well-formed trace such that

— it contains actions by the same thread,
— it begins with a txbegin action, and
— only its last action can be a committed or an aborted action.

The *status* of a transaction T is

— *committed* if it ends with a committed action;
— *aborted* if it ends with an aborted action;
— *pending* if it ends with a txcommit or a txabort action; and
— *live*, in all other cases.

A transaction T is *visible* if it contains a txcommit action and *completed* if it is either committed or aborted. An aborted transaction is *end-aborted* if it is visible, *self-aborted* if it is contains a txabort action, and *mid-aborted* otherwise. A pending transaction is *commit-pending* if it ends with a txcommit action and *abort-pending* otherwise. We denote the set of all transactions in τ by $\mathsf{tx}(\tau)$ and use the following self-explanatory notation for various subsets of transactions: $\mathsf{completed}(\tau)$, $\mathsf{committed}(\tau)$, $\mathsf{aborted}(\tau)$, $\mathsf{endaborted}(\tau)$, $\mathsf{selfaborted}(\tau)$, $\mathsf{midaborted}(\tau)$, $\mathsf{visible}(\tau)$, $\mathsf{pending}(\tau)$, $\mathsf{compending}(\tau)$, $\mathsf{abpending}(\tau)$, and $\mathsf{live}(\tau)$.

We say that an action $\varphi \in \tau$ is *transactional* if $\varphi \in T$ for some transaction $T \in \tau$ and *nontransactional* otherwise. We use the notation for transactions shown in Figure 3.

We specify the behavior of a TM implementation by the set of possible interactions it can have with programs.

Definition 3.4. A *TM* $\mathcal{T}$ *is a set of well-formed histories that is prefix-closed and closed under renaming action identifiers—for instance, if a history* H *is in* $\mathcal{T}$*, then any history equivalent to* H *is also in* $\mathcal{T}$*.*

In the definition of *transactional memories,* we require prefix-closure to take into account incomplete program executions.

3.1 An Atomic TM

We define the correctness of a TM implementation by relating its history set to that of an *abstract* implementation, whose behavior it has to simulate; in this context, we call the original implementation *concrete.* In this section, we give an example $\mathcal{T}_{\mathsf{atomic}}$ of an abstract TM that formalizes the intuitive expectations of a programmer: `atomic` blocks actually execute atomically, and methods called by aborted transactions have no effect. We start by defining more precisely what we mean by the atomic execution of atomic blocks, using the following notion of *noninterleaved* histories.

Definition 3.5. A well-formed history H is *complete* if all transactions in it are completed. A well-formed history H is *noninterleaved* if actions by any two transactions do not overlap: if $H = H_1\ (\_, t, \mathsf{txbegin})\ H_2\ (\_, t', \mathsf{txbegin})\ H_3$, where H_2 does not contain txbegin actions, then either H_2 contains a $(\_, t, \mathsf{committed})$ or a $(\_, t, \mathsf{aborted})$ action, or there are no actions by thread t in H_3.

Note that a noninterleaved history does not have to be complete. For example,

$$(\_, t, \mathsf{txbegin})\ (\_, t, \mathsf{OK})\ (\_, t, \mathsf{call}\ o.f(\_))$$
$$(\_, t', \mathsf{txbegin})\ (\_, t', \mathsf{OK})\ (\_, t', \mathsf{call}\ o.f(\_))\ (\_, t', \mathsf{ret}(\_)\ o.f)$$

is a noninterleaved history. In fact, the history set $\mathcal{T}_{\mathsf{atomic}}$ that we are about to define contains only noninterleaved histories, but some of them are incomplete. This is because programs in our language may produce traces with such histories (e.g., when due to a loop inside, an atomic block does not terminate). Hence, we need $\mathcal{T}_{\mathsf{atomic}}$ to allow these histories for it to be useful when formally defining the semantics of the programming language in Section 4.

We define $\mathcal{T}_{\mathsf{atomic}}$ in such a way that the changes made by a live, aborted, or abort-pending transaction are invisible to other transactions. However, there is no such certainty in the treatment of a commit-pending transaction: the TM implementation might have already reached a point at which it is decided that the transaction will commit. Then the transaction is effectively committed, and its operations may affect other transactions (Guerraoui and Kapalka 2011). To account for this, when defining $\mathcal{T}_{\mathsf{atomic}}$, we consider every possible completion of each commit-pending transaction in a history to either a committed or an aborted one. Formally, a history H^c is a *completion* of a well-formed history H if

— H is a subsequence of H^c,
— H^c is well formed,
— $H^c|_{\neg H}$ contains only committed or aborted actions, and
— H^c has no pending transactions.

We denote the set of all completions of H by $\mathsf{complete}(H)$.

To define $\mathcal{T}_{\mathsf{atomic}}$, we also need to know the intended semantics of operations on transactional objects. We describe the semantics for an object $o \in \mathsf{Obj}$ by fixing all sequences of actions on o that are considered correct when executed by a sequential program. More precisely, a *sequential specification* of an object o is a set of well-formed histories $[\![o]\!]$ such that

— $[\![o]\!]$ is prefix-closed;
— $[\![o]\!]$ is closed under renaming action identifiers;

— each $H \in [\![o]\!]$ consists of alternating call and ret actions on o, starting from a call action, where every ret is by the same thread as the preceding call; and
— $[\![o]\!]$ is insensitive to thread identifiers: for any $H \in [\![o]\!]$, changing the thread identifier in a call-ret pair of adjacent actions in H yields a history in $[\![o]\!]$.

For example, $[\![o]\!]$ for a register object o would consist of histories where each read method invocation returns the value written by the latest preceding write method invocation (or the default value if there is none).

Using sequential specifications for all objects, we now define when a noninterleaved history H respects the object semantics. Let $H(i)$ be a call or ret action on an object o. We say that $H(i)$ is *legal* in H if $H'|_o \in [\![o]\!]$, where H' is the history obtained from H by projecting $H|_i$ on all actions by committed transactions and the transaction containing $H(i)$. A noninterleaved history H is *legal* if all call and ret actions in H are legal. In this definition, we check every action separately to make sure that the return values are consistent with object specifications not only inside committed but also aborted and live transactions. We now let $\mathcal{T}_{\mathsf{atomic}}$ be the set of all noninterleaved histories that can be completed to a legal history:

$$\mathcal{T}_{\mathsf{atomic}} = \{H \in \mathsf{WfHistory} \mid \exists H^c \in \mathsf{complete}(H).\, H^c \text{ is legal and non-interleaved}\}.$$

We say that a TM is *atomic* if it is a subset of $\mathcal{T}_{\mathsf{atomic}}$. It is easy to check that $\mathcal{T}_{\mathsf{atomic}}$ is a prefix-closed set of well-formed noninterleaved histories.

For example, consider the history H shown later in Figure 5, and assume the expected semantics of read and write operations. Clearly, $H \notin \mathcal{T}_{\mathsf{atomic}}$, because H is not a noninterleaved history. However, the history $S = T_1T_3T_2T_4T_5$ is complete and legal: the writes to y by the aborted transactions T_3 and T_3 are ignored when checking the legality of the read from y in T_5. Hence, $S \in \mathcal{T}_{\mathsf{atomic}}$. As another example, consider the transactions shown later in Figure 7, and let $S' = T_1T_3T_4T_5$. A completion of S' where T_4 becomes committed is a noninterleaved and legal. Hence, $S' \in \mathcal{T}_{\mathsf{atomic}}$. However, $S'' = T_1T_3T_2T_4T_5$ has no legal completion: no matter how we complete T_4, we cannot justify the read of 0 from y in T_5. Hence, $S'' \notin \mathcal{T}_{\mathsf{atomic}}$.

4 SEMANTICS OF THE PROGRAMMING LANGUAGE AND OBSERVATIONAL REFINEMENT

In this section, we define the semantics of our programming language (i.e., the set of traces that computations of programs produce). The semantics comes in two flavors:

— *Semantics without rollback.* When a transaction is aborted, local variables are not rolled back to their initial values, and the values written to them by the transaction can thus be observed by the following nontransactional code.
— *Semantics with rollback.* When a transaction is aborted, local variables are rolled back to the values they had at its start, and the values written to them by the transaction cannot be observed by the following nontransactional code.

The two semantics are needed to characterize STMS and TMS, respectively. In the following, we subscript definitions used to define the semantics by RB and noRB, and use "X" to range over these.

A *state* of a program records the values of all of its variables: $s \in \mathsf{State} = \mathsf{Var} \to \mathbb{Z}$. The semantics of a program $P = C_1 \parallel \ldots \parallel C_m$ is given by the set of well-formed traces $[\![P, \mathcal{T}]\!]_{\mathsf{X}}(s) \subseteq \mathsf{WfTraces}$ it produces when executed with a TM $\mathcal{T}$ from an initial state s. We define this set in two stages. First, we define the set of traces $[\![P]\!]_{\mathsf{X}}(s) \subseteq \mathsf{WfTraces}$ that a program can produce when executed from s with the behavior of the TM unrestricted—for instance, considering all possible values, the TM can return to object method invocations and allowing transactions to commit or abort arbitrarily.

We then compute the set of traces produced by P when executed with a TM $\mathcal{T}$ by selecting those traces that interact with the TM in a way consistent with $\mathcal{T}$:

$$[\![P, \mathcal{T}]\!]_X(s) = \{\tau \mid \tau \in [\![P]\!]_X(s) \wedge \mathsf{history}(\tau) \in \mathcal{T}\}. \tag{1}$$

The set $[\![P]\!]_X(s)$ is itself defined in two stages. First, we define a set $\mathsf{Tr}(P) \subseteq \mathsf{WfTraces}$ 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 $\mathsf{Tr}(P)$ contains all possible interleavings of paths in the graphs of C_t, $t \in \mathsf{ThreadID}$, starting from their initial nodes. The set $\mathsf{Tr}(P)$ is a superset of all traces that can actually be executed—for example, if a thread executes the command

$$x := 1;\ \texttt{if}\,(x = 1)\,\{\,y := 1\,\}\,\texttt{else}\,\{\,y := 2\,\}, \tag{2}$$

where x, y are local variables, then $\mathsf{Tr}(P)$ will contain a trace where $y := 2$ is executed instead of $y := 1$. To filter out such nonsensical traces, we *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 $\mathsf{eval}_X : \mathsf{State} \times \mathsf{WfTraces} \to \mathcal{P}(\mathsf{State}) \cup \{\lightning\}$ that, given an initial state and a well-formed 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 error state $\lightning$ if the trace contains a `fault` action. Thus,

$$[\![P]\!]_X(s) = \{\tau \in \mathsf{Tr}(P) \mid \mathsf{eval}_X(s, \tau) \neq \emptyset\}. \tag{3}$$

It is the definition of the eval_X function that mandates that local variables be rolled back upon a transaction abort (when $X = \mathsf{RB}$) or not (when $X = \mathsf{noRB}$).

We next formally define the trace set $\mathsf{Tr}(P)$ (Section 4.1) and the evaluation function eval_X (Section 4.3).

4.1 The Trace Set $\mathsf{Tr}(P)$

The function $\mathsf{Tr}(\cdot)$ in Figure 4 maps programs to traces they may produce. It is defined using an auxiliary function $\mathsf{Tr}'(\cdot)$, which yields the set of all interleavings of traces produced by the threads constituting P. The set produced by $\mathsf{Tr}'(\cdot)$ may contain traces that are not well formed (e.g., because they contain duplicate identifiers or continue beyond a `fault` command). This is resolved by defining $\mathsf{Tr}(P)$ as the intersection of the set $\mathsf{Tr}'(P)$ with the set of all well-formed traces. We also take the prefix-closure of $\mathsf{Tr}'(P)$ to account for incomplete program computations, as well as those in which the scheduler preempts a thread forever.

The function $\mathsf{Tr}'_t(\cdot)$ maps a sequential command executed by thread t to the traces it may produce. $\mathsf{Tr}'_t(c)$ returns a singleton set with the action corresponding to the primitive command c (recall that primitive commands execute atomically). $\mathsf{Tr}'_t(C_1; C_2)$ concatenates all possible traces corresponding to C_1 with those corresponding to C_2. The set of traces 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 traces for a loop is defined by considering all possible unfoldings of the loop body, including the case in which it never executes. Again, we record branching decisions using assume actions.

The set of traces of a method invocation $x := f(e)$ includes both traces 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

$$
\begin{aligned}
&\mathsf{Tr}(P) && = \mathsf{prefix}(\mathsf{Tr}'(P)) \cap \mathsf{WfTraces}\\
&\mathsf{Tr}'(C_1 \parallel \ldots \parallel C_m) && = \bigcup\{\mathsf{interleave}(\tau_1,\ldots,\tau_m) \mid \forall t.\, 1 \le t \le m \implies \tau_t \in \mathsf{Tr}'_t(C_t)\}\\
&\mathsf{Tr}'_t(c) && = \{(\_, t, c)\}\\
&\mathsf{Tr}'_t(C_1; C_2) && = \{\tau_1\,\tau_2 \mid \tau_1 \in \mathsf{Tr}'_t(C_1) \wedge \tau_2 \in \mathsf{Tr}'_t(C_2)\}\\
&\mathsf{Tr}'_t(\texttt{if}\,(b)\,\{C_1\} && = \{(\_, t, \mathsf{assume}(b))\,\tau_1 \mid \tau_1 \in \mathsf{Tr}'_t(C_1)\}\\
&\quad \texttt{else}\,\{C_2\}) && \quad \cup \{(\_, t, \mathsf{assume}(\neg b))\,\tau_2 \mid \tau_2 \in \mathsf{Tr}'_t(C_2)\}\\
&\mathsf{Tr}'_t(\texttt{while}\,(b)\,\{C\}) && = \{\tau_1\,\tau_2 \ldots \tau_{2n}\,(\_, t, \mathsf{assume}(\neg b)) \mid\\
&&& \qquad n \in \mathbb{N} \wedge \forall j.\, 1 \le j \le n \implies \tau_{2j-1} = (\_, t, \mathsf{assume}(b)) \wedge \tau_{2j} \in A'(C)t\,\}\\
&&& \quad \cup \{(\_, t, \mathsf{assume}(\neg b))\}\\
&\mathsf{Tr}'_t(x := o.f(e)) && = \{(\_, t, \mathsf{assume}(e = n))\,(\_, t, \mathsf{call}\ o.f(n))\\
&&& \qquad (\_, t, \mathsf{ret}(n')\ o.f)\,(\_, t, x := n') \mid n, n' \in \mathbb{Z}\}\\
&&& \quad \cup \{(\_, t, \mathsf{assume}(e = n))\,(\_, t, \mathsf{call}\ o.f(n))\,(\_, t, \mathsf{aborted}) \mid n \in \mathbb{Z}\}\\
&\mathsf{Tr}'_t(\texttt{abort}) && = \{(\_, t, \mathsf{txabort})\,(\_, t, \mathsf{aborted})\}\\
&\mathsf{Tr}'_t(x := \texttt{atomic}\,\{C\}) && = \{(\_, t, \mathsf{txbegin})\,(\_, t, \mathsf{aborted})\,(\_, t, x := \mathsf{aborted})\}\\
&&& \quad \cup \{(\_, t, \mathsf{txbegin})\,(\_, t, \mathsf{OK})\,\tau\,(\_, t, \mathsf{aborted})\,(\_, t, x := \mathsf{aborted}) \mid\\
&&& \qquad \tau\,(\_, t, \mathsf{aborted})\,\tau' \in \mathsf{Tr}'_t(C) \wedge (\_, t, \mathsf{aborted}) \notin \tau\}\\
&&& \quad \cup \{(\_, t, \mathsf{txbegin})\,(\_, t, \mathsf{OK})\,\tau\,(\_, t, \mathsf{txcommit})\,(\_, t, r)\,(\_, t, x := r) \mid\\
&&& \qquad \tau \in \mathsf{Tr}'_t(C) \wedge (\_, t, \mathsf{aborted}) \notin \tau \wedge (r = \mathsf{committed} \vee r = \mathsf{aborted})\}
\end{aligned}
$$

Fig. 4. The definition of $\mathsf{Tr}(P)$. A trace $\tau \in \mathsf{interleave}(\tau_1, \ldots, \tau_m)$ 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 $t \in \{1, \ldots, m\}$. The function $\mathsf{prefix}(\cdot)$ prefix-closes a given set of traces.

return n' when called with the parameter n. Such infeasible choices are filtered out at the following stages of the semantics definition: the former when defining $[\![P]\!]_{\mathsf{X}}(s)$ in (3) by the semantics of assume and the latter when defining $[\![P, \mathcal{T}]\!]_{\mathsf{X}}(s)$ in (1) by selecting the traces from $[\![P]\!]_{\mathsf{X}}(s)$ that interact with the TM correctly. The trace set of $x := \texttt{atomic}\,\{C\}$ contains the trace in which the transaction aborts immediately after it is invoked, the traces in which C is aborted in the middle of its execution, and those in which C executes until completion and then the transaction commits or aborts.

4.2 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 noted in Section 2, 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

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

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 (this allows c to be nondeterministic). For example, an assignment command $x := g$ has the following semantics:

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

We define the semantics of assume commands following the informal explanations given in Section 3: for example,

$$[\![\mathsf{assume}(x = n)]\!](q) = \begin{cases} \{q\}, & \text{if } q(x) = n; \\ \emptyset, & \text{otherwise.} \end{cases} \tag{4}$$

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

We lift functions $[\![c]\!]$ to full states by keeping the variables that c is not allowed to access unmodified. 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.

4.3 Evaluation of Traces

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

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

Note that eval does not change the state s as a result of TM interface actions, as their return values are assigned to local variables by separate actions introduced when generating $\mathsf{Tr}(P)$. We now lift eval to traces; the definition of this lifting is different for the semantics with and without rollback.

Semantics without rollback. In this case, the effects of actions inside an aborted transaction on the program state are visible to the following actions. Hence, the result of evaluating a trace τ from a state s is given by the following $\mathsf{eval}_{\mathsf{noRB}}(s, \tau)$, which composes the effects on the program state of all actions in τ, including those inside aborted transactions:

$$\mathsf{eval}_{\mathsf{noRB}} : \mathsf{State} \times \mathsf{WfTraces} \to \mathcal{P}(\mathsf{State}) \cup \{\lightning\};$$

$$\mathsf{eval}_{\mathsf{noRB}}(s, \tau) = \begin{cases} \{s\}, & \tau = \varepsilon; \\ \lightning, & \tau = \tau'\varphi \text{ and } \exists s'.\, s' \in \mathsf{eval}_{\mathsf{noRB}}(s, \tau') \wedge \mathsf{eval}(s', \varphi) = \lightning; \\ \{s'' \mid \exists s'.\, s' \in \mathsf{eval}_{\mathsf{noRB}}(s, \tau') \wedge s'' \in \mathsf{eval}(s', \varphi)\}, & \\ & \tau = \tau'\varphi \text{ and } \neg\exists s'.\, s' \in \mathsf{eval}_{\mathsf{noRB}}(s, \tau') \wedge \mathsf{eval}(s', \varphi) = \lightning. \end{cases}$$

Semantics with rollback. In this case, the effects of actions inside an aborted transaction on the program state are ignored when evaluating the following actions, to model local variable rollback. To this end, the following definition of $\mathsf{eval}_{\mathsf{RB}}(s, \tau)$ removes the contents of actions inside aborted transactions before evaluating the trace. However, this poses the risk that we may consider traces including invalid sequences of actions inside aborted transactions. To mitigate this, $\mathsf{eval}_{\mathsf{RB}}(s, \tau)$ additionally evaluates every prefix of τ and checks that it yields a nonempty set of states:

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

Discussion. The preceding definitions allow us to define $[\![P]\!]_{\mathsf{X}}(s)$ as the set of those traces from $\mathsf{Tr}(P)$ that can be evaluated from s without getting stuck, as formalized by (3). Note that this enables the semantics of assume defined by (4) to filter out traces that make branching decisions inconsistent with program states. For example, consider again the program (2). The set $\mathsf{Tr}(P)$ 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 $[\![P]\!]_{\mathsf{X}}(s)$.

The semantics $[\![P, \mathcal{T}_{\mathsf{atomic}}]\!]_{\mathsf{X}}(s)$, defined by (1) for $\mathcal{T}_{\mathsf{atomic}}$ from Section 3.1, represents the intuitive expectations of the programmer about our programming language: `atomic` blocks execute without interleaving (but may not complete), and methods called by aborted transactions have no effect.

4.4 Observational Refinement

Informally, a concrete TM $\mathcal{T}_C$ observationally refines an abstract TM $\mathcal{T}_A$ if every user-observable behavior of a program using $\mathcal{T}_C$ can be reproduced if the program uses $\mathcal{T}_A$. This allows the programmer to reason about the behavior of a program using the intuitive semantics formalized by the abstract TM (e.g., $\mathcal{T}_{\mathsf{atomic}}$ from Section 3.1) while knowing that the conclusions will carry over to the program using the concrete TM. The formal definition of observational refinement depends on which aspects of program behavior we consider user observable. In this article, given a trace τ of a program, we consider observable whether τ ends with `fault` or not and, in the latter case, the sequence of nontransactional actions in τ. The rationale is that a user can always observe the program crashing, even when inside a transaction, and otherwise we assume input-output actions to be nontransactional.[2]

Definition 4.1. Well-formed traces τ and τ' are *observationally equivalent*, written $\tau \sim \tau'$, if

(i) $\tau \neq \_(\_,\_,\mathtt{fault}) \Rightarrow (\tau' \neq \_(\_,\_,\mathtt{fault}) \wedge \tau|_{\neg\mathsf{trans}} = \tau'|_{\neg\mathsf{trans}})$ and
(ii) $\tau = \_(\_,\_,\mathtt{fault}) \Rightarrow \tau' = \_(\_,\_,\mathtt{fault})$.

Note that $\sim$ is an equivalence relation.

Definition 4.2. Given $\mathsf{X} \in \{\mathsf{RB}, \mathsf{noRB}\}$, we let $\mathcal{T}_C \preceq_{\mathsf{X}} \mathcal{T}_A$ if

$$\forall P.\, \forall s.\, \forall \tau \in [\![P, \mathcal{T}_C]\!]_{\mathsf{X}}(s).\, \exists \tau' \in [\![P, \mathcal{T}_A]\!]_{\mathsf{X}}(s).\, \tau' \sim \tau.$$

If $\mathcal{T}_C \preceq_{\mathsf{noRB}} \mathcal{T}_A$, we say that $\mathcal{T}_C$ *observationally refines* $\mathcal{T}_A$ *under the semantics without rollback*. If $\mathcal{T}_C \preceq_{\mathsf{RB}} \mathcal{T}_A$, then $\mathcal{T}_C$ *observationally refines* $\mathcal{T}_A$ *under the semantics with rollback*.

If $\mathcal{T}_C \preceq_{\mathsf{X}} \mathcal{T}_A$, then the absence of faulting traces in $[\![P, \mathcal{T}_A]\!]_{\mathsf{X}}(s)$ implies the absence of such traces in $[\![P, \mathcal{T}_C]\!]_{\mathsf{X}}(s)$. Hence, the programmer can establish the safety of P assuming $\mathcal{T}_A$ instead of $\mathcal{T}_C$. The programmer can similarly establish properties of nontransactinoal actions of P and can also indirectly reason about the behavior of, for example, committed transactions by recording the required information in local variables and checking them in the following nontransactional code. However, since our notion of observations excludes actions performed inside live transactions

[2]With a few exceptions (Welc et al. 2008; Spear et al. 2008), most systems do not allow input-output actions inside transactions.

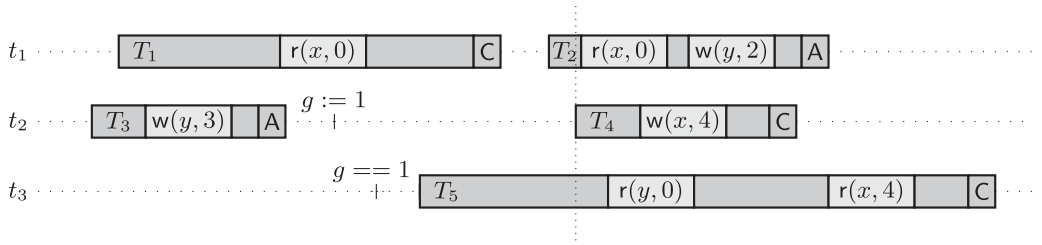


Fig. 5. Example of a history and a trace for explaining the requirement of real-time order preservation. x and y are register objects initialized to 0. $\mathsf{w}(y, n)$ and $\mathsf{r}(x, m)$ denote operations writing value n to y and reading value m from x, respectively. We denote committed or aborted status of transactions by C and A, respectively. g is a global variable.

other than faulting, the programmer cannot make any conclusions about the properties of such actions.

5 CONSISTENCY CONDITIONS AND MAIN RESULTS

We now consider three TM consistency conditions and relate them to observational refinement: TMS (Doherty et al. 2013); STMS, which we propose; and the opacity relation (Attiya et al. 2013a), a variant of opacity (Guerraoui and Kapalka 2008). We start by formalizing these conditions in our setting.[3]

As is common in consistency conditions for shared memory concurrency, such as linearizability (Herlihy and Wing 1990), a crucial building block in the definitions of the preceding consistency conditions is the following notion of the *real-time order*, which captures the order between nonoverlapping transactions in a history.

Definition 5.1. Let $\psi = (\_, t, \_)$ and $\psi' = (\_, t', \_)$ be two actions in a history H. Then ψ *is before* ψ' *in the transactional real-time order* in H, denoted by $\psi \prec_H \psi'$, if $H = H_1\psi H_2 H_2'\psi' H_3$ 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 transactional 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 transactional real-time order* in H, denoted by $T \prec_H T'$, if $T \prec_H T'(1)$.

For example, in history H in Figure 5, $T_1 \prec_H T_2$ and $T_3 \prec_H T_5$; however, $\neg(T_1 \prec_H T_5)$. It is easy to verify that $\prec_H$ is a partial order.

Definition 5.2. The transactional real-time order of a history H is *preserved* in a history S, denoted by $H \sqsubseteq_{\mathsf{RT}} S$, if

$$\forall \psi, \psi'. (\psi \in S \iff \psi \in H) \wedge (\psi \prec_H \psi' \Rightarrow \psi \prec_S \psi').$$

For example, considering again the history H in Figure 5, we can see that $H \sqsubseteq_{\mathsf{RT}} T_1T_3T_2T_4T_5$. It is easy to verify that $\sqsubseteq_{\mathsf{RT}}$ is a partial order. For brevity, in the following, we use the term *real-time order* instead of *transactional real-time order*.

[3]TMS was originally formulated in an operational manner, using I/O automata; here we present a more abstract definition appropriate for our goals.

5.1 The STMS Relation

The consistency conditions that we define relate a concrete TM $\mathcal{T}_C$ and an abstract TM $\mathcal{T}_A$. For every history $H \in \mathcal{T}_C$, the conditions require the existence of one or more histories $S \in \mathcal{T}_A$ *justifying* the TM behavior in H. Our intention is for the STMS relation $\mathcal{T}_C \sqsubseteq_{\mathsf{STMS}} \mathcal{T}_A$ to be equivalent to observational refinement under the semantics without rollback: $\mathcal{T}_C \preceq_{\mathsf{noRB}} \mathcal{T}_A$. Hence, STMS picks justifying histories so that a trace τ of a program P with a history $H \in \mathcal{T}_C$ could be transformed into an observationally equivalent trace τ' of P with a history $S \in \mathcal{T}_A$ justifying H. STMS picks justifications in two ways, which can respectively be used to transform τ into τ' in the cases when τ contains a fault inside a live transaction or not (see Definition 4.1). We start by defining the STMS requirements that allow performing the transformation in the latter case.

Consider first a complete history $H \in \mathcal{T}_C$. In this case, STMS requires the existence of a history $S \in \mathcal{T}_A$ such that $H \sqsubseteq_{\mathsf{RT}} S$. In particular, S has to contain the same transactions as H. This is necessary for STMS to imply observational refinement under the semantics without rollback: since all transactions in H are completed, the nontransactional code following the corresponding `atomic` blocks in P is aware of the return values obtained inside these transactions, even if they aborted. For example, even if the transaction by thread 1 in Figure 1 is aborted, the value of local variable `pos`, which depends on the transactional object `Top`, is assigned to the global variable `g`.

Thus, to convert a trace τ of P with the history $H \in \mathcal{T}_C$ into a trace τ' of P with the same nontransactional actions, but with a history $S \in \mathcal{T}_A$, the history S has to match the return values inside the transactions of H. The relative positions of actions by different transactions may generally differ in H and S, but we require that the real-time order of H be preserved in S. This is also necessary for observational refinement. As illustrated in Figure 5, if $T_3 \prec_H T_5$, then in between T_3 completing and T_5 starting in the trace τ, threads t_2 and t_3 may execute nontransactional code and can thus communicate using global variables, such as g. Preserving the real-time order of H in S ensures that this communication can be preserved when transforming τ into τ'.

We now consider the general case of a history $H \in \mathcal{T}_C$, which requires us to deal with live and commit-pending transactions in H. In this case, STMS requires the existence of a history $S \in \mathcal{T}_A$ such that $H^c \sqsubseteq_{\mathsf{RT}} S$ for some history H^c that can be obtained from H via the following transformation: all live and abort-pending transactions and any number of commit-pending transactions are discarded, and the remaining commit-pending transactions are completed with committed actions. The intuition is that since a live or an abort-pending transaction has not made an attempt to commit, its actions should not affect completed transactions, which influence the nontransactional code following them in a program. A commit-pending transaction may or may not have effectively committed, and thus its actions may or may not affect completed transactions (Section 3.1).

To perform the preceding transformations on H, we use the following auxiliary definitions and operations:

- A history H^c is a *commit-completion* of a well-formed history H if it is a completion of H and there exists a history H' comprised only of committed actions such that $H^c = HH'$. Let $\mathsf{cendcomplete}(H)$ be the set of all commit-completions of H.
- Let $\mathsf{rempending}(H)$ be the set of histories obtained from H by removing any number of its commit-pending transactions and all its abort-pending transactions: we have $H' \in \mathsf{rempending}(H)$ if and only if
  - H' is a subsequence of H,
  - $\mathsf{tx}(H) \setminus \mathsf{pending}(H) = \mathsf{tx}(H') \setminus \mathsf{pending}(H')$, and
  - $\mathsf{pending}(H') \subseteq \mathsf{compending}(H)$.

Given the preceding definitions, a part of the STMS relation between $\mathcal{T}_C$ and $\mathcal{T}_A$ requires

$$\forall H \in \mathcal{T}_C.\ \exists H' \in \mathsf{rempending}(H|_{\neg\mathsf{live}}).\ \exists H^c \in \mathsf{cendcomplete}(H').\ \exists S \in \mathcal{T}_A.\ H^c \sqsubseteq_{\mathsf{RT}} S. \tag{5}$$

The other part of STMS considers every response action ψ in a history $H = H_1\psi H_2 \in \mathcal{T}_A$ that is not a committed or an aborted action. It then requires the existence of a certain history $S_\psi \in \mathcal{T}_A$ justifying the outcome of ψ. As we show, this allows transforming a trace τ of P that has the history H and contains a fault in the transaction of ψ into a trace τ' of P that has the history S_ψ and contains a fault in the same transaction. The history $S_\psi \in \mathcal{T}_A$ includes the transaction of ψ and some of the transactions from $H_1\psi$. Somewhat counterintuitively, these transactions may exclude some of the committed transactions in $H_1\psi$ maximal in the real-time order, and may include some of such aborted transactions with their status changed to committed: the response ψ is given as if the latter transactions have taken effect and the former have not. Thus, S_ψ creates a "virtual world" that describes the view of ψ on the TM state. This view may be different from that of a response action ψ' in another transaction, as STMS allows $S_{\psi'}$ to be different from S_ψ.

To formalize this part of STMS, we first introduce several auxiliary definitions. A transaction $T \in \mathsf{tx}(H)$ is *maximal* in H if it is not followed by another transaction in the real-time order: $\neg\exists T'.\, T' \in \mathsf{tx}(H).\, T \prec_H T'$. We denote the set of all maximal transactions of H by $\mathsf{maxtx}(H)$. The following notion of a *possible past* of a history $H = H_1\psi$ defines all sets of transactions from H that can form its justification S_ψ.

Definition 5.3. A well-formed history $H_\psi = H_1'\psi$ is a *possible past* of a well-formed history $H = H_1\psi$, where ψ is a response action that it is not a committed or an 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 completed or commit-pending transactions in H:

$$\mathsf{tx}(H_\psi) \subseteq \{\mathsf{txof}(\psi, H)\} \cup \mathsf{committed}(H) \cup \mathsf{aborted}(H) \cup \mathsf{compending}(H).$$

In particular, H_ψ does not contain abort-pending transactions or live transactions, except that of ψ.

(iii) For every transaction $T \in H_\psi$, the history H_ψ includes all transactions preceding T in the real-time order in H:

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

(iv) A maximal transaction in H_ψ cannot be mid- or self-aborted:

$$(\mathsf{midaborted}(H_\psi) \cup \mathsf{selfaborted}(H_\psi)) \cap \mathsf{maxtx}(H_\psi) = \emptyset.$$

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

We explain the definition using the history $H_1\psi$ of the trace shown in Figure 6; one of its possible pasts H_ψ consists of the transactions T_1, T_3, T_4, and T_5. According to (i) and (ii), the transaction of ψ (T_5 in Figure 6) is always included into any possible past, and other live transactions as well as abort-pending ones are excluded: since they have not made an attempt to commit, they should not have an effect on ψ. We are allowed to select which of the remaining transactions to include into S_ψ subject to (iii): if we include a transaction T, then we also have to include all transactions preceding it in the real-time order. For example, since T_4 and T_5 are included in H_ψ, so are T_1 and T_3. This condition is necessary for STMS to imply observational refinement. To illustrate, consider a trace τ of a program P with the history H that has a fault inside the transaction T_5. As illustrated in Figure 6, in between T_3 aborting and T_5 starting in τ, thread t_2 can communicate to thread t_3 some facts about the behavior of T_3 (e.g., using a global variable g). Then to preserve the behavior of T_5 when transforming τ into a trace τ' of P with a history from $\mathcal{T}_A$ constructed from the transactions in H_ψ, we also have to preserve the behavior of T_3. Hence, T_3 has to be included into H_ψ: informally, the views of T_3 and T_5 on the TM behavior have to be consistent.

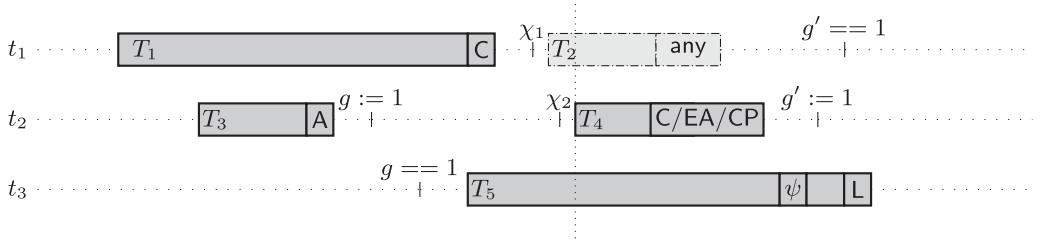


Fig. 6. Transactions T_1, T_3, T_4, and T_5 form one possible past of the history H of the trace shown. Allowed status of transactions in H is denoted as follows: committed, C; aborted, A; end-aborted, EA; commit-pending, CP; live, L. The transaction T_5 executes only primitive actions after ψ in the trace. χ_1 and χ_2 are nontransactional actions by threads t_1 and t_2, respectively.

As the next step in constraining the desired justification S_ψ of H_ψ, we change the status of all commit-pending and maximal aborted transactions in H_ψ to committed, as we define in the following. To ensure that the resulting history is well formed, condition (iv) allows including mid- or self-aborted transactions into H_ψ only when they are not maximal (i.e., only when their inclusion is required by condition (iii)).

Definition 5.4. We let $\mathsf{maxcom}(H)$ denote the history obtained from H by making all maximal aborted transactions in H committed: $|\mathsf{maxcom}(H)| = |H|$ and

$$\mathsf{maxcom}(H)(i) = (\text{if } (H(i) = (a, t, \text{aborted}) \wedge \mathsf{txof}(H(i), H) \in \mathsf{maxtx}(H)), \\ \text{then } (a, t, \text{committed}) \text{ else } H(i)).$$

Definition 5.5. The set of *completed possible pasts* of a well-formed history $H_1\psi$ is

$$\mathsf{cSTMSpast}(H_1\psi) = \{H^c_\psi \mid \exists H'_1, H''_1.\, H'_1\psi \in \mathsf{STMSpast}(H_1\psi) \wedge \\ H''_1 = \mathsf{maxcom}(H'_1) \wedge H^c_\psi \in \mathsf{cendcomplete}(H''_1\psi)\}.$$

For example, one completed possible past of the history in Figure 6 consists of the transactions T_1, T_3, T_4, and T_5, with the status of T_4 changed to committed if it was previously aborted or commit-pending. We can now give a complete definition of STMS, strengthening (5).

Definition 5.6. A history H is in the *STMS relation* with TM $\mathcal{T}$, denoted $H \sqsubseteq_{\mathsf{STMS}} \mathcal{T}$, if

(i) $\exists H' \in \mathsf{rempending}(H|_{\neg\mathsf{live}}).\, \exists H^c \in \mathsf{cendcomplete}(H').\, \exists S \in \mathcal{T}.\, H^c \sqsubseteq_{\mathsf{RT}} S$; and
(ii) for every response action ψ such that it is not a committed or an aborted action and $H = H_1\psi H_2$ for some H_1 and H_2, we have $\exists H^c_\psi \in \mathsf{cSTMSpast}(H_1\psi).\, \exists S_\psi \in \mathcal{T}.\, H^c_\psi \sqsubseteq_{\mathsf{RT}} S_\psi$.

A TM $\mathcal{T}_C$ is in the *STMS relation* with a TM $\mathcal{T}_A$, denoted by $\mathcal{T}_C \sqsubseteq_{\mathsf{STMS}} \mathcal{T}_A$, if

$$\forall H \in \mathcal{T}_C.\, H \sqsubseteq_{\mathsf{STMS}} \mathcal{T}_A.$$

Thus, part (i) in this definition checks the behavior of transactions in H excluding live ones; part (ii) then covers live transactions. Even though our definition of completed possible past of a history H allows us to perform unexpected transformations on H, the resulting definition of STMS nevertheless validates observational refinement. This is because our programming language does not allow accessing global variables inside transactions. Hence, when constructing S_ψ from H, we can change, for example, the status of T_4 from aborted to committed: 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

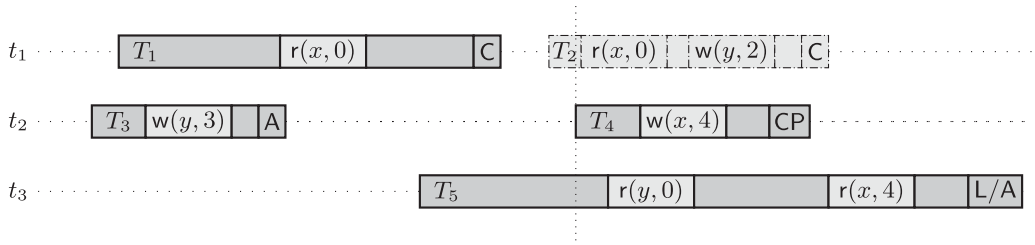


Fig. 7. Histories illustrating the definitions of STMS and TMS.

the change when replacing the concrete TM by an abstract one in observational refinement. For similar reasons, we can exclude T_2 from S_ψ, even if it is committed.

We illustrate the definition of the STMS relation using the histories in Figure 7. We take $\mathcal{T}_{\mathsf{atomic}}$ as the abstract TM (Section 3.1), with the expected semantics of read and write operations. We now argue that $H \sqsubseteq_{\mathsf{STMS}} \mathcal{T}_{\mathsf{atomic}}$ for the history H depicted in the figure assuming transaction T_5 is live. To satisfy the conditions in Definition 5.6(i), we can take as H' and H^c the subsequence of H consisting of transactions T_1, T_2, T_3 and as S the history $S = T_1T_3T_2 \in \mathcal{T}_{\mathsf{atomic}}$. Another option is to take as H' the subsequence of H consisting of transactions T_1, T_2, T_3, and T_4; as H^c, we can take the history $H\psi$, where ψ is a committed action completing T_4 to a committed transaction T_4'. Then we can satisfy the definition by letting $S = T_1T_3T_2T_4' \in \mathcal{T}_{\mathsf{atomic}}$.

To establish the conditions in Definition 5.6(ii), we need to consider every prefix $H_1\psi$ of H ending with a response action ψ that is not an aborted or committed action. We explain the case when ψ is the response for the read of x by T_5. For this ψ, a completed possible past H^c_ψ contains T_1, T_3, T_5 and the committed transaction T_4', obtained from T_4 as explained previously. To satisfy the definition, we can then take $S_\psi = T_1T_3T_4'T_5 \in \mathcal{T}_{\mathsf{atomic}}$. Note that we have to include T_4' into H^c_ψ, because the read of x by T_5 returns the value written by T_4.

Consider now the history H in Figure 7, assuming that T_5 is aborted. We argue that $\neg(H \sqsubseteq_{\mathsf{STMS}} \mathcal{T}_{\mathsf{atomic}})$. Since T_4 is the only transaction that writes to x and T_5 reads 4 from x, Definition 5.6(i) requires us to find a complete history $S \in \mathcal{T}_{\mathsf{atomic}}$ that consists of transactions T_1, T_2, T_3, T_5 and the transaction T_4' obtained by committing T_4. However, this is not possible: for T_5 to read 4 from x, it has to follow T_4' in S; for T_2 to read 0 from x, it must appear before T_4'; but then since T_2 writes 2 to y, T_5 cannot read 0 from y.

5.2 The TMS Relation

TMS (Doherty et al. 2013) is a weaker consistency condition than STMS. As we show, the TMS relation $\mathcal{T}_C \sqsubseteq_{\mathsf{TMS}} \mathcal{T}_A$ is equivalent to the observational refinement under the semantics with rollback: $\mathcal{T}_C \preceq_{\mathsf{RB}} \mathcal{T}_A$. Since this semantics does not allow nontransactional code to observe the behavior inside aborted transactions, TMS is obtained from STMS by excluding such transactions from the checks that it performs. Formally, this is achieved by removing aborted transactions from completed possible pasts of Definition 5.5 and from the history H^c in Definition 5.6(i).

Definition 5.7. The set of *completed visible possible pasts* of a well-formed history H is

$$\mathsf{cTMSpast}(H) = \{(H'|_{\neg\mathsf{aborted}}) \mid H' \in \mathsf{cSTMSpast}(H)\}.$$

Definition 5.8. A history H is in the *TMS relation* with TM $\mathcal{T}$, denoted by $H \sqsubseteq_{\mathsf{TMS}} \mathcal{T}$, if

(i) $\exists H' \in \mathsf{rempending}(H|_{\neg\mathsf{live}}).\ \exists H^c \in \mathsf{cendcomplete}(H').\ \exists S \in \mathcal{T}.\ H^c|_{\neg\mathsf{aborted}} \sqsubseteq_{\mathsf{RT}} S$; and

C1 A TM $\mathcal{T}$ is *closed under removing some aborted, abort-pending and live 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

$$\mathsf{committed}(H') = \mathsf{committed}(H) \wedge \mathsf{compending}(H') = \mathsf{compending}(H) \wedge$$
$$\mathsf{aborted}(H') \subseteq \mathsf{aborted}(H) \wedge \mathsf{abpending}(H') \subseteq \mathsf{abpending}(H) \wedge \mathsf{live}(H') \subseteq \mathsf{live}(H).$$

C2 A TM $\mathcal{T}$ is *closed under completing pending transactions* if whenever $H \in \mathcal{T}$, we have $\mathsf{complete}(H) \cap \mathcal{T} \neq \emptyset$.

C3 A TM $\mathcal{T}$ is *closed under adding self-aborts* if whenever $H \in \mathcal{T}$ and there is a live transaction in H by a thread t which ends with a response action, we have $H^a \in \mathcal{T}$ for any well-formed history $H^a = H\,(\_, t, \mathsf{txabort})$.

C4 A TM $\mathcal{T}$ is *closed under completing pending actions* if whenever $H \in \mathcal{T}$, we also have $HH' \in \mathcal{T}$ for some history H' comprised of response actions such that the last action of every thread in HH' is a response action.

C5 A TM $\mathcal{T}$ is *closed under removing self-aborts* if whenever $H = H_1 \psi H_2 \in \mathcal{T}$, where $\psi = (\_, t, \mathsf{txabort})$ and H_2 does not contain any actions by thread t, we have $H_1 H_2 \in \mathcal{T}$.

C6 A TM $\mathcal{T}$ is *closed under removing final responses* if whenever $H \in \mathcal{T}$, we also have $H' \in \mathcal{T}$ for any well-formed history H' which is a subsequence of H such that any action $\psi \in H|_{\neg H'}$ is a response that is not a committed action.

C7 A TM $\mathcal{T}$ is *closed under immediate aborts* if whenever $H \in \mathcal{T}$ and $H_a \in \mathsf{addoneab}(H)$, we also have $H_a \in \mathcal{T}$. Here $H_a \in \mathsf{addoneab}(H)$ if H_a is well-formed and for some H', H'' we have $H = H'H''$, $H_a = H'\,(\_, t, \mathsf{txbegin})\,(\_, t, \mathsf{aborted})\,H''$ and

$$H' = \epsilon \vee H' = \_\,(\_, \_, \mathsf{committed}) \vee H' = \_\,(\_, \_, \mathsf{aborted}). \tag{6}$$

Fig. 8. Closure properties on TMs. C1, C2, and C5 through C7 are required of abstract TMs, and C3, C4 of concrete TMs.

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

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

$$\forall H \in \mathcal{T}_C.\, H \sqsubseteq_{\mathsf{TMS}} \mathcal{T}_A.$$

Consider again the history H in Figure 7 assuming that T_5 is aborted. Earlier we argued that $\neg(H \sqsubseteq_{\mathsf{STMS}} \mathcal{T}_{\mathsf{atomic}})$. We now argue that $H \sqsubseteq_{\mathsf{TMS}} \mathcal{T}_{\mathsf{atomic}}$. To satisfy the conditions in Definition 5.8(i), we take as H' the history H and as H^c the history $H\psi$, where the action ψ completes T_4 to a committed transaction T'_4. Then H^c aborted consists of transactions T_1, T_2, and T'_4, and we can satisfy the definition by letting $S = T_1T_2T'_4 \in \mathcal{T}_{\mathsf{atomic}}$. We illustrate Definition 5.8(ii) for the case when the action ψ is the response for the read of x by T_5. When explaining STMS, we argued that for this ψ, a completed possible past consists of T_1, T_3,T_5 and the committed transaction T'_4. We can thus satisfy the definition by choosing H^c_ψ so that it consists of T_1, T_5, and T'_4 and letting $S_\psi = T_1T'_4T_5 \in \mathcal{T}_{\mathsf{atomic}}$.

We now prove that under a certain condition on abstract TMs, STMS implies TMS. This and other conditions on TMs used in this article are listed in Figure 8; we introduce them as needed. All of them have the form of *closure properties* on the set of histories defining a TM behavior; some are required of abstract TMs and some of concrete ones. The former are satisfied by $\mathcal{T}_{\mathsf{atomic}}$, which formalizes the intuitive expectations of a programmer (Section 3.1). Informally, closure property C1, which we use here, requires that actions by aborted, abort-pending, and live transactions do not

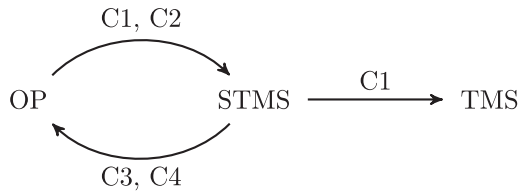


Fig. 9. Relationships between STMS, TMS, and opacity.

$$H \xrightarrow{\text{Definition 5.6(i)}} H^c \sqsubseteq_{\mathsf{RT}} S \xrightarrow{\text{Proposition 5.9}} H^c|_{\neg\mathsf{aborted}} \sqsubseteq_{\mathsf{RT}} S|_{\neg\mathsf{aborted}}$$

$$H = H_1\psi H_2 \xrightarrow{\text{Definition 5.6(ii)}} H^c_\psi \sqsubseteq_{\mathsf{RT}} S_\psi \xrightarrow{\text{Proposition 5.9}} H^c_\psi|_{\neg\mathsf{aborted}} \sqsubseteq_{\mathsf{RT}} S_\psi|_{\neg\mathsf{aborted}}$$

$$\downarrow \text{C1}$$

$$S_\psi|_{\neg\mathsf{aborted}} \in \mathcal{T}_A$$

Fig. 10. Some of the constructions in the proof of Theorem 5.10.

affect other transactions.[4] We summarize the relationships that we establish in this article between various consistency conditions and the closure properties used for this in Figure 9.

Our proof of the relationship between STMS and TMS uses the following proposition, which ensures that projection does not introduce new real-time order relationships.

Proposition 5.9. $\forall H_1, H_2, H.\ H_1 \sqsubseteq_{\mathsf{RT}} H_2 \Rightarrow H_1|_{\neg H} \sqsubseteq_{\mathsf{RT}} H_2|_{\neg H}$.

Theorem 5.10. *Consider TMs $\mathcal{T}_C$ and $\mathcal{T}_A$ such that the latter satisfies closure property C1. Then $\mathcal{T}_C \sqsubseteq_{\mathsf{STMS}} \mathcal{T}_A \Rightarrow \mathcal{T}_C \sqsubseteq_{\mathsf{TMS}} \mathcal{T}_A$.*

Proof. We illustrate the proof in Figure 10. Take $H \in \mathcal{T}_C$. By Definition 5.6(i), there exist (i) a history $H' \in \mathsf{rempending}(H|_{\neg\mathsf{live}})$, (ii) a history $H^c \in \mathsf{cendcomplete}(H')$, and (iii) a history $S \in \mathcal{T}_A$ such that $H^c \sqsubseteq_{\mathsf{RT}} S$. By Proposition 5.9, $H^c \sqsubseteq_{\mathsf{RT}} S$ implies $H^c|_{\neg\mathsf{aborted}} \sqsubseteq_{\mathsf{RT}} S|_{\neg\mathsf{aborted}}$. Since $\mathcal{T}_A$ satisfies C1, from $S \in \mathcal{T}_A$ we get $S|_{\neg\mathsf{aborted}} \in \mathcal{T}_A$. Together with (i) and (ii), this establishes the requirements of Definition 5.8(i).

Let ψ be a response action in H that is not a committed or an aborted action, and let $H = H_1\psi H_2$. By Definition 5.6(ii), there exist histories $H^c_\psi \in \mathsf{cSTMSpast}(H_1\psi)$ and $S_\psi \in \mathcal{T}_A$ such that $H^c_\psi \sqsubseteq_{\mathsf{RT}} S_\psi$. Since $S_\psi \in \mathcal{T}_A$ and $\mathcal{T}_A$ satisfies C1, we have $S_\psi|_{\neg\mathsf{aborted}} \in \mathcal{T}_A$. Since $H^c_\psi \in \mathsf{cSTMSpast}(H_1\psi)$, by Definition 5.7 we get $H^c_\psi|_{\neg\mathsf{aborted}} \in \mathsf{cTMSpast}(H_1\psi)$. Finally, by Proposition 5.9, we get $H^c_\psi|_{\neg\mathsf{aborted}} \sqsubseteq_{\mathsf{RT}} S_\psi|_{\neg\mathsf{aborted}}$. Together with $S_\psi|_{\neg\mathsf{aborted}} \in \mathcal{T}_A$, this establishes the requirements of Definition 5.8(ii). □

5.3 The Opacity Relation

The opacity relation (Attiya et al. 2013a) requires us to justify the behavior of all transactions in a history of the concrete TM, including aborted and live ones, by a single history of the abstract TM.

[4]Note that this rules out *early release*, which allows live transactions to read values written by other live transactions (Herlihy et al. 2003; Marathe et al. 2005).

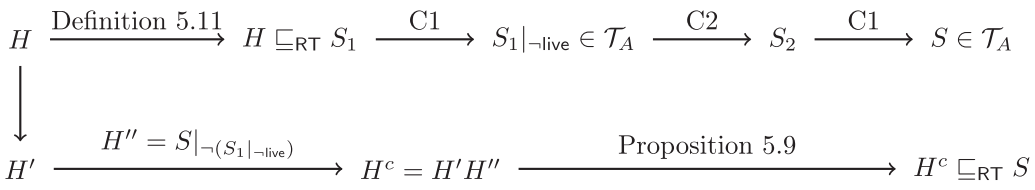


Fig. 11. Some of the constructions in Part I of the proof of Theorem 5.12.

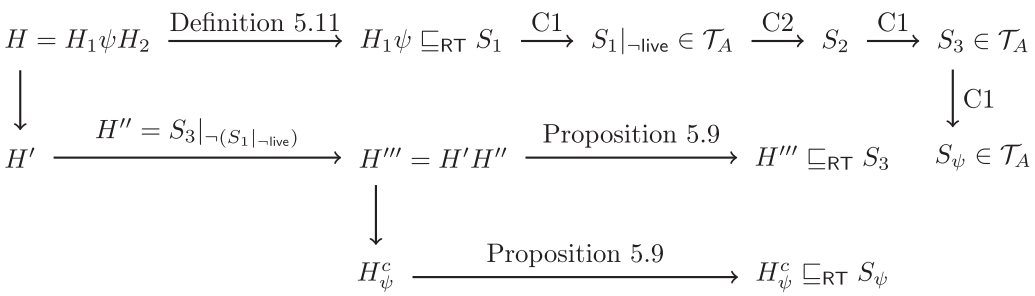


Fig. 12. Some of the constructions in Part II of the proof of Theorem 5.12.

Definition 5.11. A TM $\mathcal{T}_C$ is in the *opacity relation* with a TM $\mathcal{T}_A$, denoted by $\mathcal{T}_C \sqsubseteq_{\mathsf{OP}} \mathcal{T}_A$, if $\forall H \in \mathcal{T}_C.\ \exists S \in \mathcal{T}_A.\ H \sqsubseteq_{\mathsf{RT}} S$.

As shown by Attiya et al. (2013a), in the case when $\mathcal{T}_A = \mathcal{T}_{\mathsf{atomic}}$ (Section 3.1), the opacity relation becomes equivalent to a well-known condition of *opacity* (Guerraoui and Kapalka 2008). Under certain conditions on abstract TMs, the opacity relation implies STMS.

Theorem 5.12. *Consider TMs $\mathcal{T}_C$ and $\mathcal{T}_A$ such that the latter satisfies C1 and C2. Then $\mathcal{T}_C \sqsubseteq_{\mathsf{OP}} \mathcal{T}_A \Rightarrow \mathcal{T}_C \sqsubseteq_{\mathsf{STMS}} \mathcal{T}_A$.*

Proof. We illustrate the proof in Figures 11 and 12. Let $H \in \mathcal{T}_C$. The proof has two parts, which respectively establish the conditions in Definition 5.6(i) and (ii).

Part I. Since $\mathcal{T}_C \sqsubseteq_{\mathsf{OP}} \mathcal{T}_A$, there is a history $S_1 \in \mathcal{T}_A$ such that $H \sqsubseteq_{\mathsf{RT}} S_1$. Since $\mathcal{T}_A$ satisfies C1, we have $S_1|_{\neg\mathsf{live}} \in \mathcal{T}_A$. Since $\mathcal{T}_A$ satisfies C2, there exists a complete history $S_2 \in \mathsf{complete}(S_1|_{\neg\mathsf{live}}) \cap \mathcal{T}_A$. Let S be the subsequence of S_2 obtained by removing the transactions that got aborted when $S_1|_{\neg\mathsf{live}}$ was completed to S_2. Since $\mathcal{T}_A$ satisfies C1, we have $S \in \mathcal{T}_A$.

Let H' be the subsequence of $H|_{\neg\mathsf{live}}$ obtained by removing the transactions that got aborted in S_2. As any abort-pending transaction in H necessarily got aborted in S_2, we have $H' \in \mathsf{rempending}(H|_{\neg\mathsf{live}})$. Let $H'' = S|_{\neg(S_1|_{\neg\mathsf{live}})}$ be the sequence of committed actions used to complete transactions in S, and let $H^c = H'H''$ (without loss of generality, we can assume that the identifiers of the actions in H' and H'' are distinct). By construction, $H^c \in \mathsf{cendcomplete}(H')$. It is easy to see that $\forall t.\ H^c|_t = S|_t$. Furthermore, when transforming H' into H^c, we place committed actions at its end and thus do not create new real-time order relationships. Using this fact and Proposition 5.9, we establish $H^c \sqsubseteq_{\mathsf{RT}} S$.

Part II. Let ψ be a response action in H that is not a committed or an aborted action, and let $H = H_1\psi H_2$. Since $\mathcal{T}_C$ is prefix-closed, we have $H_1\psi \in \mathcal{T}_C$. Then since $\mathcal{T}_C \sqsubseteq_{\mathsf{OP}} \mathcal{T}_A$, there is a history $S_1 \in \mathcal{T}_A$ such that $H_1\psi \sqsubseteq_{\mathsf{RT}} S_1$. Since $\mathcal{T}_A$ satisfies C1, we have $S_1|_{\neg\mathsf{live}} \in \mathcal{T}_A$. Since $\mathcal{T}_A$ satisfies C2, there exists a complete history $S_2 \in \mathsf{complete}(S_1|_{\neg\mathsf{live}}) \cap \mathcal{T}_A$. Let S_3 be the subsequence of S_2 obtained by removing all transactions that got aborted when $S_1|_{\neg\mathsf{live}}$ was completed to S_2. Since $\mathcal{T}_A$ satisfies C1, we have $S_3 \in \mathcal{T}_A$.

Let H' be the subsequence of $(H_1\psi)|_{\neg\mathsf{live}}$ obtained by removing the transactions that got aborted in S_2; note that this removes all abort-pending transactions from $(H_1\psi)|_{\neg\mathsf{live}}$. Let $H'' = S_3|_{\neg(S_1|_{\neg\mathsf{live}})}$ be the sequence of committed actions used to complete transactions in S_3, and let $H''' = H'H''$ (without loss of generality, we can assume that the identifiers of the actions in H' and H'' are distinct). Like in the previous part of the proof, we have $H''' \sqsubseteq_{\mathsf{RT}} S_3$. Let $H^c_\psi \in \mathsf{WfHistory}$ be the well-formed history such that

(i) H^c_ψ is a subsequence of H''',
(ii) $\mathsf{tx}(H'''|_{\neg H^c_\psi}) \subseteq \mathsf{aborted}(H''')$,
(iii) $\mathsf{maxtx}(H^c_\psi) \cap \mathsf{aborted}(H^c_\psi) = \emptyset$, and
(iv) $\forall T_1, T_2.\, T_1 \prec_H T_2 \wedge T_2 \in \mathsf{tx}(H^c_\psi) \Rightarrow T_1 \in \mathsf{tx}(H^c_\psi)$.

It is easy to check that such H^c_ψ exists and is unique. Note that H^c_ψ was obtained from H''' by removing some aborted transactions. Let S_ψ be the subsequence of S_3 obtained by removing the same transactions. Then $S_\psi \in \mathcal{T}_A$, since $\mathcal{T}_A$ satisfies C1. Furthermore, Proposition 5.9 ensures that $H^c_\psi \sqsubseteq_{\mathsf{RT}} S_\psi$. By the construction of H^c_ψ, we have that $H^c_\psi|_{\neg H''} \in \mathsf{STMSpast}(H)$ and $H^c_\psi \in \mathsf{cendcomplete}(H^c_\psi|_{\neg H''})$. Furthermore, since H^c_ψ does not contain maximal aborted transactions, we also have $\mathsf{maxcom}(H^c_\psi|_{\neg H''}) = H^c_\psi|_{\neg H''}$. From this, it follows that $H^c_\psi \in \mathsf{cSTMSpast}(H_1\psi)$, which completes the proof. □

In general, STMS does not imply the opacity relation. Consider, for example, the history shown in Figure 7, assuming that T_5 is a live transaction. We have already argued that $H \sqsubseteq_{\mathsf{STMS}} \mathcal{T}_{\mathsf{atomic}}$ (Section 5.1). However, $\neg(H \sqsubseteq_{\mathsf{OP}} \mathcal{T}_{\mathsf{atomic}})$. This is the case for essentially the same reasons as $\neg(H \sqsubseteq_{\mathsf{STMS}} \mathcal{T}_{\mathsf{atomic}})$, assuming that T_5 is aborted in H, as we discussed before. We now show that under certain conditions on *concrete* TMs, STMS does imply opacity. The following lemma establishes that this relationship always holds for complete histories.

Lemma 5.13. *Let H be a complete history and $\mathcal{T}$ a TM such that $H \sqsubseteq_{\mathsf{STMS}} \mathcal{T}$. Then for some history $S \in \mathcal{T}$, we have $H \sqsubseteq_{\mathsf{RT}} S$.*

Proof. By Definition 5.6(i), there exist histories $H' \in \mathsf{rempending}(H|_{\neg\mathsf{live}})$, $H^c \in \mathsf{cendcomplete}(H')$, and $S \in \mathcal{T}$ such that $H^c \sqsubseteq_{\mathsf{RT}} S$. Since H is complete, we have $H' = H$ and $\mathsf{cendcomplete}(H') = \mathsf{cendcomplete}(H) = \{H\}$. Hence, $H^c = H$ so that $H \sqsubseteq_{\mathsf{RT}} S$. □

The next theorem establishes that STMS implies opacity in the general case, provided that the concrete TM satisfies C3 and C4. The former property requires the concrete TM to always allow the client to explicitly abort a transaction. The latter property is a liveness property similar to lock-freedom (Herlihy and Shavit 2008): it requires that the concrete TM eventually respond to client requests, provided the client does not issue new ones. Informally, these properties ensure that STMS imply opacity, because they allow completing all live transactions in any history produced by the concrete TM. Then Definition 5.6(i) forces the TM to give a single justification for all transactions in the history, making the flexibility allowed by Definition 5.6(ii) unnecessary. The only way in which a TM may satisfy STMS, but not opacity, is by giving to a live transaction (e.g., T_5 in Figure 7) a view of the TM state that is inconsistent with that of other transactions, then never responding to a request to commit or abort the live transaction. We also rely on closure properties C5 and C6 on abstract TMs, which mirror C3 and C4. Like C1, these are satisfied by TMs where actions by live and aborted transactions do not affect other transactions.

Theorem 5.14. *Let $\mathcal{T}_C$ be a TM that satisfies C3 and C4, and let $\mathcal{T}_A$ be a TM that satisfies C5 and C6. Then $\mathcal{T}_C \sqsubseteq_{\mathsf{STMS}} \mathcal{T}_A \Rightarrow \mathcal{T}_C \sqsubseteq_{\mathsf{OP}} \mathcal{T}_A$.*

Proof. Take $H \in \mathcal{T}_C$. Since $\mathcal{T}_C$ satisfies C4, there exists a history $HH' \in \mathcal{T}_C$, which extends H with a sequence of response actions such that every request action in HH' has a matching response. $\mathcal{T}_C$ also satisfies C3, and thus there exists a complete history $HH'H'' \in \mathcal{T}_C$ such that H'' contains a txabort action for every live transaction in HH'. As $\mathcal{T}_C$ satisfies C4, there exists a history $H^c = HH'H''H''' \in \mathcal{T}_C$ such that H''' aborts all abort-pending transactions introduced by H''. In particular, H^c is a complete history.

Since $\mathcal{T}_C \sqsubseteq_{\mathsf{STMS}} \mathcal{T}_A$, we get $H^c \sqsubseteq_{\mathsf{STMS}} \mathcal{T}_A$. Because H^c is a complete history, Lemma 5.13 ensures that for some history $S^c \in \mathcal{T}_A$ we have $H^c \sqsubseteq_{\mathsf{RT}} S^c$. Let S be the history produced by removing all actions in S^c added to H while completing it to H^c: $S = S^c|_{\neg H'H''H'''}$. Note that $S \in \mathcal{T}_A$ because $S^c \in \mathcal{T}_A$ and $\mathcal{T}_A$ satisfies C6 and C5. Thus, by Proposition 5.9, we get $H \sqsubseteq_{\mathsf{RT}} S$, which implies the required. □

5.4 Statements of the Main Results

Assuming certain closure properties on abstract TMs, we establish that the STMS relation is equivalent to observational refinement under the semantics without rollback and the TMS relation is equivalent to observational refinement under the semantics with rollback.

Theorem 5.15. *Let $\mathcal{T}_C$ and $\mathcal{T}_A$ be TMs:*

(i) $\mathcal{T}_C \sqsubseteq_{\mathsf{STMS}} \mathcal{T}_A \Longrightarrow \mathcal{T}_C \preceq_{\mathsf{noRB}} \mathcal{T}_A$.
(ii) If $\mathcal{T}_A$ satisfies C1 and C2, then $\mathcal{T}_C \preceq_{\mathsf{noRB}} \mathcal{T}_A \Longrightarrow \mathcal{T}_C \sqsubseteq_{\mathsf{STMS}} \mathcal{T}_A$.

Theorem 5.16. *Let $\mathcal{T}_C$ and $\mathcal{T}_A$ be TMs:*

(i) If $\mathcal{T}_A$ satisfies C7, then $\mathcal{T}_C \sqsubseteq_{\mathsf{TMS}} \mathcal{T}_A \Longrightarrow \mathcal{T}_C \preceq_{\mathsf{RB}} \mathcal{T}_A$.
(ii) If $\mathcal{T}_A$ satisfies C1 and C2, then $\mathcal{T}_C \preceq_{\mathsf{RB}} \mathcal{T}_A \Longrightarrow \mathcal{T}_C \sqsubseteq_{\mathsf{TMS}} \mathcal{T}_A$.

The closure property C7 in Theorem 5.16 is a technical requirement, allowing us to add empty aborted transactions to histories of an abstract TM. Constraint (6) is needed so that this operation transforms noninterleaved histories into noninterleaved ones, and the closure property can be satisfied by $\mathcal{T}_{\mathsf{atomic}}$ (Section 3.1).

As part of proving the preceding theorems, we also prove the next result, showing that the opacity relation is sufficient for observational refinement without assuming any closure properties on TMs.

Theorem 5.17. $\forall \mathsf{X} \in \{\mathsf{RB}, \mathsf{noRB}\}.\ \mathcal{T}_C \sqsubseteq_{\mathsf{OP}} \mathcal{T}_A \Rightarrow \mathcal{T}_C \preceq_{\mathsf{X}} \mathcal{T}_A$.

6 SUFFICIENCY PROOFS

In this section, we prove Theorems 5.15(i) and 5.16(i). Let us fix a program P and a state s. To prove the theorems, we need to transform a trace $\tau \in [\![P]\!]_{\mathsf{X}}(s)$ with a history $H \in \mathcal{T}_C$ into an observationally equivalent trace $\tau' \in [\![P]\!]_{\mathsf{X}}(s)$ with a history $S \in \mathcal{T}_A$. This transformation is done differently depending on whether τ contains a fault inside a live transaction or not, and these cases respectively exploit the two clauses in Definitions 5.6 and 5.8. We start by presenting the key lemmas used in the two transformations (Sections 6.1 and 6.2).

6.1 Sufficiency of the Opacity Relation

Consider the following notion of trace equivalence, strengthening the one in Definition 4.1.

Definition 6.1. Well-formed traces τ and τ' are *strongly equivalent*, denoted $\tau \approx \tau'$, if

$$(\tau|_{\neg\mathsf{trans}} = \tau'|_{\neg\mathsf{trans}}) \wedge (\forall t \in \mathsf{ThreadID}.\ \tau|_t = \tau'|_t).$$

The following lemma (holding for any $\mathsf{X} \in \{\mathsf{RB}, \mathsf{noRB}\}$) is key in the trace transformation using Definitions 5.6(i) and 5.8(i), which match histories using the real-time order relation. It shows that a trace $\tau_H \in [\![P]\!]_{\mathsf{X}}(s)$ with a history H can be transformed into a strongly equivalent trace τ_S with a given history $S \in [\![P]\!]_{\mathsf{X}}(s)$ such that $H \sqsubseteq_{\mathsf{RT}} S$.

Lemma 6.2.

$$\forall P.\, \forall s.\, \forall H, S \in \mathsf{WfHistory}.\, H \sqsubseteq_{\mathsf{RT}} S \Rightarrow$$
$$(\forall \tau_H \in [\![P]\!]_{\mathsf{X}}(s).\, \mathsf{history}(\tau_H) = H \Rightarrow \exists \tau_S \in [\![P]\!]_{\mathsf{X}}(s).\, \mathsf{history}(\tau_S) = S \wedge \tau_H \approx \tau_S).$$

Before proving the lemma, we note some of its consequences. Let us formulate a notion of observational refinement between TMs induced by the trace equivalence in Definition 6.1.

Definition 6.3. Given $\mathsf{X} \in \{\mathsf{RB}, \mathsf{noRB}\}$, we let $\mathcal{T}_C \preceq_{\mathsf{X}} \mathcal{T}_A$ if

$$\forall P.\, \forall s.\, \forall \tau \in [\![P, \mathcal{T}_C]\!]_{\mathsf{X}}(s).\, \exists \tau' \in [\![P, \mathcal{T}_A]\!]_{\mathsf{X}}(s).\, \tau' \approx \tau.$$

Then Lemma 6.2 implies the following theorem, showing that the opacity relation is sufficient for this notion of observational refinement.

Theorem 6.4. $\forall \mathsf{X} \in \{\mathsf{RB}, \mathsf{noRB}\}.\, \mathcal{T}_C \sqsubseteq_{\mathsf{OP}} \mathcal{T}_A \Rightarrow \mathcal{T}_C \preceq_{\mathsf{X}} \mathcal{T}_A$.

Since strong equivalence (Definition 6.1) implies observational equivalence (Definition 4.1), Theorem 5.17 is a straightforward corollary of Theorem 6.4. Strong equivalence of Definition 6.1 requires preserving not only nontransactional actions and faults but also the behavior of all live transactions in the trace, whether faulting or not. Since at most one live transaction may fault in a trace, and we assume that input-output actions are nontransactional, strong equivalence thus preserves aspects of program behavior that are unobservable to the program user in practice. Hence, even though we use strong equivalence in intermediate results, our final goal is establishing a link between TM consistency conditions and observational refinement between TMs based on the weaker Definition 4.1 (we discuss this further in Section 8).

We now proceed to prove Lemma 6.2. The next lemma gives the key step in this proof; unlike Lemma 6.2, it transforms arbitrary traces, not necessarily those produced by P.

Lemma 6.5 (Rearrangement).

$$\forall H, S \in \mathsf{WfHistory}.\, H \sqsubseteq_{\mathsf{RT}} S \Rightarrow$$
$$(\forall \tau_H \in \mathsf{WfTraces}.\, \mathsf{history}(\tau_H) = H \Rightarrow \exists \tau_S \in \mathsf{WfTraces}.\, \mathsf{history}(\tau_S) = S \wedge \tau_H \approx \tau_S).$$

Proof. Consider $H, S \in \mathsf{WfHistory}$ and $\tau_H \in \mathsf{WfTraces}$ such that $H \sqsubseteq_{\mathsf{RT}} S$ and $\mathsf{history}(\tau_H) = H$. Note that $|H| = |S|$. To obtain the desired trace τ_S, we inductively construct a sequence of traces $\tau^i \in \mathsf{WfTraces}$, $i = 0..|S|$ with histories $H^i = \mathsf{history}(\tau^i) \in \mathsf{WfHistory}$ such that

$$H^i\lfloor_i = S\lfloor_i; \quad H^i \sqsubseteq_{\mathsf{RT}} S; \quad \tau_H \approx \tau^i. \tag{7}$$

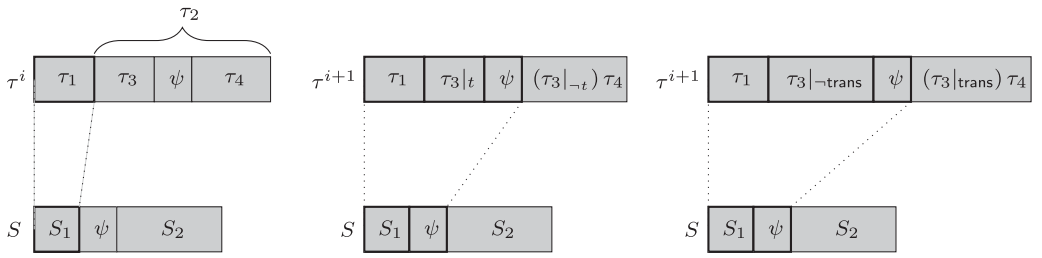
We then let $\tau_S = \tau^{|S|}$ so that $\tau_H \approx \tau^{|S|}$ and

$$\mathsf{history}(\tau^{|S|}) = H^{|S|} = H^{|S|}\lfloor_{|S|} = S\lfloor_{|S|} = S,$$

as required. Note that the condition $H^i \sqsubseteq_{\mathsf{RT}} S$ in (7) is not used to establish the required properties of τ_S; we add it so that the induction goes through.

We start the construction of the sequence of traces τ^i with $\tau^0 = \tau_H$ so that $H^0 = H$ and all requirements in (7) hold trivially. Assume that a trace $\tau^i \in \mathsf{WfTraces}$ satisfying (7) has been constructed. We show that there is a history $H^{i+1} \in \mathsf{WfHistory}$ and a trace $\tau^{i+1} \in \mathsf{WfTraces}$ such that

$$\mathsf{history}(\tau^{i+1}) = H^{i+1}; \quad H^{i+1}\lfloor_{i+1} = S\lfloor_{i+1}; \quad H^{i+1} \sqsubseteq_{\mathsf{RT}} S; \quad \tau^i \approx \tau^{i+1}.$$



(a) Trace τ^i and history S (b) Case of $\psi \neq (\_, t, \text{txbegin})$ (c) Case of $\psi = (\_, t, \text{txbegin})$

Fig. 13. Illustration of the transformations performed in the proof of Lemma 6.5.

Let $S = S_1\psi S_2$, where $|S_1| = i$. By (7), $\text{history}(\tau^i)\downarrow_i = H^i\downarrow_i = S\downarrow_i = S_1$. Thus, for some traces τ_1 and τ_2, we have $\tau^i = \tau_1\tau_2$, where τ_1 is the minimal prefix of τ^i such that $\text{history}(\tau_1) = S_1$ (see Figure 13(a)). We also have $H^i \sqsubseteq_{\text{RT}} S$, and hence S is a permutation of H^i preserving the real-time order (Definition 5.2). Since $\text{history}(\tau_1) = S_1$ and $\text{history}(\tau^i) = S_1\psi S_2$, for some traces τ_3 and τ_4, we have

$$\tau_2 = \tau_3\psi\tau_4, \qquad \tau^i = \tau_1\tau_2 = \tau_1\tau_3\psi\tau_4.$$

Let $\psi = (\_, t, \_)$. We note that since $\sqsubseteq_{\text{RT}}$ preserves the order of actions by the same thread and $\text{history}(\tau_1) = S_1$, we have $\text{history}(\tau_3|_t) = \varepsilon$. We consider two cases, depending on whether $\psi = (\_, t, \text{txbegin})$ or not.

Case I: $\psi \neq (\_, t, \text{txbegin})$. Let $\tau^{i+1} = \tau_1\,(\tau_3|_t)\,\psi\,(\tau_3|_{\neg t})\,\tau_4$ and $H^{i+1} = \text{history}(\tau^{i+1})$ (see Figure 13(b)). Intuitively, τ^{i+1} is obtained from $\tau^i = \tau_1\tau_3\psi\tau_4$ by moving all actions in τ_3 performed by thread t, together with ψ, to the position right after τ_1.

Since $\text{history}(\tau_1) = S_1$, $\text{history}(\tau_3|_t) = \varepsilon$, and $|S_1| = i$, we get

$$H^{i+1}\downarrow_{i+1} = (\text{history}(\tau_1\,(\tau_3|_t)\,\psi\,(\tau_3|_{\neg t})\,\tau_4))\downarrow_{i+1} = S_1\psi = S\downarrow_{i+1},$$

as required. We also have

$$\tau^{i+1}|_t = (\tau_1\,(\tau_3|_t)\,\psi\,(\tau_3|_{\neg t})\,\tau_4)|_t = (\tau_1|_t)\,(\tau_3|_t)\,\psi\,(\tau_4|_t) = (\tau_1\tau_3\psi\tau_4)|_t = \tau^i|_t;$$
$$\tau^{i+1}|_{\neg t} = (\tau_1\,(\tau_3|_t)\,\psi\,(\tau_3|_{\neg t})\,\tau_4)|_{\neg t} = (\tau_1|_{\neg t})\,(\tau_3|_{\neg t})\,(\tau_4|_{\neg t}) = (\tau_1\tau_3\psi\tau_4)|_{\neg t} = \tau^i|_{\neg t}.$$

Hence, for any thread t', we have $\tau^i|_{t'} = \tau^{i+1}|_{t'}$ and $H^{i+1}|_{t'} = H^i|_{t'} = S|_{t'}$. If a committed or an aborted action precedes a txbegin action in H^{i+1}, but not in H^i, then the precedence also holds in S. Thus, $H^{i+1} \sqsubseteq_{\text{RT}} S$.

Since $\psi \neq (\_, t, \text{txbegin})$ and $\text{history}(\tau_3|_t) = \varepsilon$, all actions performed by t in the subtrace τ_3 of τ^i are transactional. Hence,

$$\tau^{i+1}|_{\neg\text{trans}} = (\tau_1\,(\tau_3|_t)\,\psi\,(\tau_3|_{\neg t})\,\tau_4)|_{\neg\text{trans}} = (\tau_1\tau_3\psi\tau_4)|_{\neg\text{trans}} = \tau^i|_{\neg\text{trans}},$$

and therefore $\tau^i \approx \tau^{i+1}$.

Case II: $\psi = (\_, t, \text{txbegin})$. Assume that the subtrace τ_3 of τ^i contains a committed or aborted action ψ'. Since $\text{history}(\tau_1) = S_1$, the action ψ' would be in S_2. This would mean that the real-time order between ψ' and ψ in H^i is not preserved in S, contradicting our assumption that $H^i \sqsubseteq_{\text{RT}} S$. Thus, for any thread $t' \neq t$, $\tau_3|_{t'}$ consists of some number of nontransactional actions followed by some number of transactional ones, and $\tau_3|_t$ does not contain any transactional actions. Motivated by these observations, we let

$$\tau^{i+1} = \tau_1\,(\tau_3|_{\neg\text{trans}})\,\psi\,(\tau_3|_{\text{trans}})\,\tau_4$$

and $H^{i+1} = \mathsf{history}(\tau^{i+1})$ (see Figure 13(c)). Intuitively, τ^{i+1} is obtained from $\tau^i = \tau_1\tau_3\psi\tau_4$ by moving all transactional actions in τ_3 to the position right before τ_4.

Since $\mathsf{history}(\tau_1) = S_1$, $\mathsf{history}(\tau_3|_{\neg\mathsf{trans}}) = \varepsilon$, and $|S_1| = i$, we get

$$H^{i+1}\downarrow_{i+1} = (\mathsf{history}(\tau_1\,(\tau_3|_{\neg\mathsf{trans}})\,\psi\,(\tau_3|_{\mathsf{trans}})\,\tau_4))\downarrow_{i+1} = S_1\psi = S\downarrow_{i+1},$$

as required.

Since for every thread $t' \neq t$, $\tau_3|_{t'}$ consists of nontransactional actions followed by transactional ones,

$$((\tau_3|_{\neg\mathsf{trans}})\,\psi\,(\tau_3|_{\mathsf{trans}}))|_{t'} = (\tau_3\psi)|_{t'}.$$

Since $\tau_3|_t$ does not contain any transactional actions, $\tau_3|_t = (\tau_3|_{\neg\mathsf{trans}})|_t$, and hence

$$((\tau_3|_{\neg\mathsf{trans}})\,\psi\,(\tau_3|_{\mathsf{trans}}))|_t = (\tau_3\psi)|_t.$$

Thus, for any t'', we have

$$\tau^{i+1}|_{t''} = (\tau_1\,(\tau_3|_{\neg\mathsf{trans}})\,\psi\,(\tau_3|_{\mathsf{trans}})\,\tau_4)|_{t''} = (\tau_1\tau_3\psi\tau_4)|_{t''} = \tau^i|_{t''},$$

and $H^{i+1}|_{t''} = H^i|_{t''} = S|_{t''}$. If a committed or an aborted action precedes a txbegin action in H^{i+1}, then it also does in H^i. Hence, $H^{i+1} \sqsubseteq_{\mathsf{RT}} S$. Finally,

$$\tau^{i+1}|_{\neg\mathsf{trans}} = (\tau_1\,(\tau_3|_{\neg\mathsf{trans}})\,\psi\,(\tau_3|_{\mathsf{trans}})\,\tau_4)|_{\neg\mathsf{trans}} = (\tau_1\tau_3\psi\tau_4)|_{\neg\mathsf{trans}} = \tau^i|_{\neg\mathsf{trans}},$$

and hence $\tau^i \approx \tau^{i+1}$. □

We also need the following lemma, which states that if two traces are strongly equivalent, then one of the traces is valid if the other is as well. The proof of the lemma relies on the restrictions on accesses to variables in Definition 3.2.

Lemma 6.6.

$$\forall\tau_H, \tau_S \in \mathsf{WfTraces}.\ \forall s.\ \tau_H \approx \tau_S \wedge \mathsf{eval}_{\mathsf{X}}(s, \tau_H) \neq \emptyset \Rightarrow \mathsf{eval}_{\mathsf{X}}(s, \tau_H) = \mathsf{eval}_{\mathsf{X}}(s, \tau_S).$$

Proof. The proof goes by case analysis depending on whether τ_H contains a `fault` action or not.

Case I. Let us assume first that τ_H, and hence τ_S, does not contain a `fault` action. We inductively construct a sequence of traces $\tau^i \in \mathsf{WfTraces}$, $i = 0..|\tau_S|$ such that

$$\tau^i\downarrow_i = \tau_S\downarrow_i;\quad \tau^i \approx \tau_S;\quad \mathsf{eval}_{\mathsf{X}}(s, \tau^i) = \mathsf{eval}_{\mathsf{X}}(s, \tau_H) \neq \emptyset. \tag{8}$$

Then for $i = |\tau_S|$, we get $\tau^i = \tau_S$, which implies the required result.

To construct the sequence of traces τ^i, we let $\tau^0 = \tau_H$ so that all requirements in (8) hold trivially. Assume now that a trace τ^i satisfying (8) has been constructed. Let $\tau_S = \tau_1\varphi\tau_2$, where $|\tau_1| = i$, and $\varphi = (\_, t, \_)$. By assumption, $\tau^i\downarrow_i = \tau_S\downarrow_i$ and $\tau^i \approx \tau_S$ so that $\tau^i|_t = \tau_S|_t$. Hence, for some traces τ_2' and τ_2'', we get $\tau^i = \tau_1\tau_2'\varphi\tau_2''$, where τ_2' does not contain any actions by thread t. Let $\tau^{i+1} = \tau_1\varphi\tau_2'\tau_2''$; then $\tau^{i+1}\downarrow_{i+1} = \tau_S\downarrow_{i+1}$.

We now show $\tau^{i+1} \approx \tau^i$. Note that $\tau^i|_{t'} = \tau^{i+1}|_{t'}$ for any thread t', as τ_2' does not contain any actions by thread t. Because $\tau^i \approx \tau_S$, we have $\tau^i|_{\neg\mathsf{trans}} = \tau_S|_{\neg\mathsf{trans}}$. We also have $\tau^i = \tau_1\tau_2'\varphi\tau_2''$ and $\tau_S = \tau_1\varphi\tau_2$. Then if φ is nontransactional, then τ_2' cannot contain any nontransactional actions. Hence, $\tau^{i+1}|_{\neg\mathsf{trans}} = \tau^i|_{\neg\mathsf{trans}}$, and thus $\tau^{i+1} \approx \tau^i$.

We next show $\mathsf{eval}_{\mathsf{X}}(s, \tau^{i+1}) = \mathsf{eval}_{\mathsf{X}}(s, \tau^i)$, which completes the proof of this case. The restrictions on accesses to variables by commands from LPcomm_t and GPcomm_t (stated in Section 2 and formalized in Section 4) imply the following.

Proposition 6.7. *Assume that* $\chi_1 = (\_, t_1, c_1)$ *and* $\chi_2 = (\_, t_2, c_2)$ *are actions by different threads and*

$$(c_1 \in \mathsf{LPcomm}_{t_1} \wedge c_2 \in \mathsf{LPcomm}_{t_2} \uplus \mathsf{GPcomm}_{t_2}) \vee$$
$$(c_1 \in \mathsf{LPcomm}_{t_1} \uplus \mathsf{GPcomm}_{t_1} \wedge c_2 \in \mathsf{LPcomm}_{t_2}).$$

Then $\mathsf{eval}_{\mathsf{X}}(s, \chi_1\chi_2) = \mathsf{eval}_{\mathsf{X}}(s, \chi_2\chi_1)$ *for any state* s.

Since τ^i is well formed, by Definition 3.2, for any action $(\_, t', c)$ in it, we have that $c \in \mathsf{LPcomm}_{t'} \uplus \mathsf{GPcomm}_{t'}$, and if $c \in \mathsf{GPcomm}_{t'}$, then the action is nontransactional. Given this and the properties of τ_2' established previously, by applying Proposition 6.7 repeatedly, we get that $\mathsf{eval}_{\mathsf{X}}(s', \varphi\tau_2') = \mathsf{eval}_{\mathsf{X}}(s', \tau_2'\varphi)$ for any state s'. Hence,

$$\mathsf{eval}_{\mathsf{X}}(s, \tau^{i+1}) = \mathsf{eval}_{\mathsf{X}}(s, \tau_1\varphi\tau_2'\tau_2'') = \mathsf{eval}_{\mathsf{X}}(s, \tau_1\tau_2'\varphi\tau_2'') = \mathsf{eval}_{\mathsf{X}}(s, \tau^i).$$

Case II. We now consider the case when $\tau_H = \tau_H'(\_, t, \mathtt{fault})$. By assumption, $\tau_H \approx \tau_S$. Hence, $\tau_H|_t = \tau_S|_t$, and because τ_S is well formed, $\tau_S = \tau_S'(\_, t, \mathtt{fault})$ and neither τ_H' nor τ_S' contains a $\mathtt{fault}$. Then $\tau_H' \approx \tau_S'$. By assumption, we also have $\mathsf{eval}_{\mathsf{X}}(s, \tau_H) \neq \emptyset$, and hence $\mathsf{eval}_{\mathsf{X}}(s, \tau_H') \neq \emptyset$ and $\mathsf{eval}_{\mathsf{X}}(s, \tau_H) = \mathtt{fault}$. Applying Case I, we get $\mathsf{eval}_{\mathsf{X}}(s, \tau_S') = \mathsf{eval}_{\mathsf{X}}(s, \tau_H') \neq \emptyset$. By the definition of $\mathsf{eval}_{\mathsf{X}}$, this implies $\mathsf{eval}_{\mathsf{X}}(s, \tau_S) = \lightning = \mathsf{eval}_{\mathsf{X}}(s, \tau_H)$. □

Proof of Lemma 6.2. Let $P = C_1 \parallel \ldots \parallel C_m$. Consider s and $H, S \in \mathsf{WfHistory}$ and $\tau_H \in [\![P]\!]_{\mathsf{X}}(s)$ such that $\mathsf{history}(\tau_H) = H$. By Lemma 6.5, there exists $\tau_S \in \mathsf{WfTraces}$ such that $\mathsf{history}(\tau_S) = S$ and $\tau_H \approx \tau_S$. It remains to show that $\tau_S \in [\![P]\!]_{\mathsf{X}}(s)$.

Since $\tau_H \in [\![P]\!]_{\mathsf{X}}(s)$, for some τ', we have $\tau_H\tau' \in \mathsf{Tr}'(P)$. This implies $(\tau_H\tau')|_t \in \mathsf{Tr}_t'(C_t)$ for any thread t. Since $\tau_H \approx \tau_S$, we have $\tau_S|_t = \tau_H|_t$, and so $(\tau_S\tau')|_t \in \mathsf{Tr}_t'(C_t)$. Then by the definition of $\mathsf{Tr}(P)$ in Figure 4, we get $\tau_S \in \mathsf{Tr}(P)$.

Since $\tau_H \in [\![P]\!]_{\mathsf{X}}(s)$, we also have $\mathsf{eval}_{\mathsf{X}}(s, \tau_H) \neq \emptyset$. Together with $\tau_H \approx \tau_S$, by Lemma 6.6, this implies $\mathsf{eval}_{\mathsf{X}}(s, \tau_S) \neq \emptyset$. But we have also established $\tau_S \in \mathsf{Tr}(P)$ so that $\tau_S \in [\![P]\!]_{\mathsf{X}}(s)$. □

6.2 The Live Transaction Insensitivity Lemma

As part of the sufficiency proofs, we need to transform a trace $\tau \in [\![P]\!]_{\mathsf{X}}(s)$ with a history $H \in \mathcal{T}_C$ into an observationally equivalent trace $\tau' \in [\![P]\!]_{\mathsf{X}}(s)$ with a history $S \in \mathcal{T}_A$. The previous section dealt with the case when H and S are related as in part (i) of Definitions 5.6 and 5.8. We now deal with the other case, when H and S are related as in part (ii). The main subtlety of this case lies in the fact that part (ii) allows justifying the behavior of a live transaction in H by a history S that contains only a subset of transactions in H, with the committed/aborted status of some of these transactions changed; this is formalized by the use of cSTMSpast and cTMSpast in Definitions 5.6(ii) and 5.8(ii), respectively. The subtle connection between H and S makes it challenging to show that a fault inside a live transaction of τ can be reproduced in τ', as required by Definition 4.1(ii).

The following lemma describes the first and foremost step of this transformation: given a trace $\tau \in [\![P]\!]_{\mathsf{X}}(s)$ with a live transaction and a history $H_\psi^c \in \mathsf{cSTMSpast}(\mathsf{history}(\tau))$, the lemma converts τ into another trace from $[\![P]\!]_{\mathsf{X}}(s)$ that contains the same live transaction but whose history is H_ψ^c. In other words, this establishes that the live transaction cannot notice changes to the set of transactions done by applying cSTMSpast. The lemma holds for both $\mathsf{X} = \mathsf{RB}$ and $\mathsf{X} = \mathsf{noRB}$, and even though the lemma deals only with cSTMSpast, it is used in the proofs of both sufficiency results.

Lemma 6.8 (Live Transaction Insensitivity). *Let* $\tau = \tau_1\psi\tau_2 \in [\![P]\!]_{\mathsf{X}}(s)$ *be a trace 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* $H_\psi^c \in \mathsf{cSTMSpast}(\mathsf{history}(\tau))$ *such that the action identifiers*

in $H^c_\psi|_{\neg\mathsf{history}(\tau)}$ *do not appear in* τ*, there exists* $\tau_\psi \in [\![P]\!]_X(s)$ *such that* $\mathsf{history}(\tau_\psi) = H^c_\psi$ *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 6. Let $\mathsf{history}(\tau) = H_1\psi$. Since $H^c_\psi \in \mathsf{cSTMSpast}(H)$, by Definition 5.5 there exist histories H_1' and H_1'' such that

$$H_1'\psi \in \mathsf{STMSpast}(H_1\psi) \ \wedge\ H_1'' = \mathsf{maxcom}(H_1') \ \wedge\ H^c_\psi \in \mathsf{cendcomplete}(H_1''\psi).$$

In the following, we use H^{cc} to denote the subsequence of H^c_ψ comprised of the committed actions that were inserted into $H_1''\psi$ to obtain H^c_ψ (i.e., $H^{cc} = H^c_\psi|_{\neg H_1''\psi}$).

Recall that for the τ in Figure 6, one possible $H_1'\psi$ consists of the transactions T_1, T_3, 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, as 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 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_1^b and τ_2^b, we have $\tau = \tau_1^b\psi^b\tau_2^b\psi\tau_2$. For the trace τ in Figure 6, ψ^b is the txbegin action of T_4. Our idea is, for every thread other than t_0, to erase all of its actions that follow the last of its transactions included into $H_1'\psi$ or its last nontransactional action preceding ψ^b, whichever is later.

Formally, for every thread t, let τ_t^I 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 6, $\tau^I_{t_1}$ and $\tau^I_{t_2}$ end with the last TM interface actions of T_1 and T_4, respectively. Similarly, let τ_t^N denote the prefix of $\tau|_t$ that ends with the last nontransactional action of t in τ_1^b, or ε if no such action exists. For example, in Figure 6, $\tau^N_{t_1}$ and $\tau^N_{t_2}$ respectively end with the actions χ_1 and χ_2, directly preceding T_2 and T_4. Let $\tau_{t_0} = \tau|_{t_0}$, and for each $t \neq t_0$, let τ_t be the longer of the traces τ_t^N and τ_t^I—i.e., τ_t^I, if $|\tau_t^N| < |\tau_t^I|$, and τ_t^N, otherwise. We define the truncated trace τ' as the subsequence of τ such that $\tau'|_t = \tau_t$ for each t. Thus, for the τ in Figure 6, 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'. The proof now proceeds in several stages.

Stage I: $\mathsf{history}(\tau') = H_1'\psi$. By the choice of τ_t^I for $t \neq t_0$, every transaction in $(H_1'\psi)|_t$ is also in τ_t^I. In addition, $\tau'|_{t_0} = \tau_{t_0}$. Hence, $H_1'\psi$ is a subsequence of $\mathsf{history}(\tau')$. We now show that every transaction in $\mathsf{history}(\tau')$ is in $H_1'\psi$. We consider three cases, depending on the thread t that the transaction is by:

— $t = t_0$. Let $T = \mathsf{txof}(\psi, H_1\psi) \in H_1'\psi$. Then by Definition 5.3(iii), we get

$$\forall T'.\, T' \prec_{H_1'\psi} T \iff T' \prec_{H_1\psi} T. \tag{9}$$

Since any transaction T' in $\mathsf{history}(\tau'|_{t_0}) = \mathsf{history}(\tau|_{t_0})$ is either T or is such that $T' \prec_{(H_1\psi)|_{t_0}} T$, (9) implies the required.

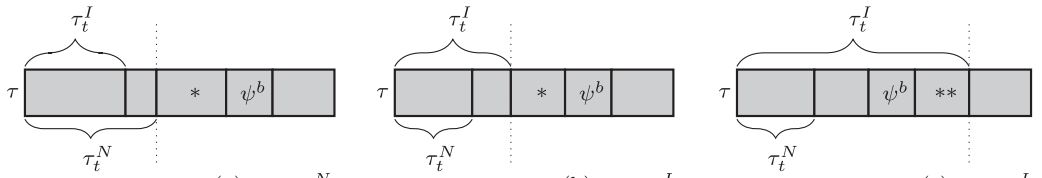


Fig. 14. Cases in the proof of Lemma 6.8. *, all actions by t are transactional; **, all actions by t come from a single transaction, started before or by ψ^b.

— $t \neq t_0$ is such that $\tau'|_t = \tau_t^I \neq \varepsilon$. Let ψ_t^I be the last action in τ_t^I, and 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 5.3(iii), we get (9). Since any transaction T' in $\mathsf{history}(\tau'|_t)$ is either T or is such that $T' \prec_{(H_1\psi)|_t} T$, (9) implies the required.
— $t \neq t_0$ is such that $\tau'|_t = \tau_t^N \neq \varepsilon$. Let χ_t^N be the last action in τ_t^N, and let $T = \mathsf{txof}(\psi^b, H_1\psi) \in H_1'\psi$. Then by Definition 5.3(iii), we get (9). Since χ_t^N 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 (9) implies the required.

This shows that $\mathsf{history}(\tau') = H_1'\psi$.

Stage II: Constructing τ_ψ such that $\mathsf{history}(\tau_\psi) = H_\psi^c$ and $\tau_\psi|_{t_0} = \tau|_{t_0}$. To construct τ_ψ from τ', we mirror the transformations of H_1' into H_1'' and H_ψ^c. Let τ'' be defined by $|\tau''| = |\tau'|$ and

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

Then $\mathsf{history}(\tau'') = H_1''\psi$. Given this and Definition 5.3(iv), which ensures that any maximal aborted transaction in $H_1''\psi$ is visible, it is easy to see that τ'' is well formed. We now construct τ_ψ from τ'' similarly to how H_ψ^c is constructed from $H_1''\psi$, by inserting actions in H^{cc} so that $\mathsf{history}(\tau_\psi) = H_\psi^c$. Since the action identifiers in $H_\psi^c|_{\neg\mathsf{history}(\tau)}$ do not appear in τ, the trace τ_ψ is well formed.

Finally, we have $\tau_\psi|_{t_0} = \tau|_{t_0}$, because $\tau'|_{t_0} = \tau|_{t_0}$; any aborted transaction in $\mathsf{history}(\tau|_{t_0})$ precedes the live transaction $\mathsf{txof}(\psi, H_1\psi)$ in the real-time order and hence is not maximal; and H^{cc} does not contain any action by t_0.

Stage III: $\tau' \in [\![P]\!]_X(s)$. 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 14. Either $\tau_t = \tau_t^N$ (a, thread t_1 in Figure 6) or $\tau_t = \tau_t^I$ (b, c). If $\tau_t = \tau_t^N$, then ψ^b has to come in τ after the end of τ_t^N. If $\tau_t = \tau_t^I$, then either ψ^b comes after the end of τ_t^I (b) or ψ^b is the last action of τ_t^I or precedes this action (c, thread t_2 in Figure 6).

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 6). 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 txbegin action by t; hence, 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 6). 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 6).

Given the aforementioned analysis and the fact that $\tau'|_{t_0} = \tau|_{t_0}$, 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, according

to the semantics of Section 4, they are not affected by the actions of other threads. Furthermore, $[\![P]\!]_{\mathsf{X}}(s)$ includes incomplete program computations. This allows us to conclude that $\tau' \in [\![P]\!]_{\mathsf{X}}(s)$.

Stage IV: $\tau_\psi \in [\![P]\!]_{\mathsf{X}}(s)$. Consider a transaction T by a thread t whose status is changed when switching from τ' to τ''. Then $t \neq t_0$ and T must be the last transaction in τ'_t. We again consider cases (a) through (c). In case (a), we have $\mathsf{history}(T) \prec_{H'_1\psi} \mathsf{txof}(\psi^b, H'_1\psi)$. Hence, the status of T is not changed when switching from τ' to τ''. In cases (b) and (c), T does not have any nontransactional actions following it. Since τ'' is well formed, T is also visible. Since the definition of $[\![P]\!]_{\mathsf{X}}(s)$ allows committing or aborting transactions arbitrarily, we conclude that $\tau'' \in [\![P]\!]_{\mathsf{X}}(s)$. For the same reason, we get $\tau_\psi \in [\![P]\!]_{\mathsf{X}}(s)$. □

6.3 Proof of Theorem 5.15(i): Sufficiency of the STMS Relation

Assume $\mathcal{T}_C \sqsubseteq_{\mathsf{STMS}} \mathcal{T}_A$. Consider $\tau \in [\![P, \mathcal{T}_C]\!]_{\mathsf{noRB}}(s)$, and let $H = \mathsf{history}(\tau)$.

Assume first that τ does not contain a `fault` action inside a live transaction. Since $\mathcal{T}_C \sqsubseteq_{\mathsf{STMS}} \mathcal{T}_A$, there exist $H' \in \mathsf{rempending}(H|_{\neg\mathsf{live}})$, $H^c \in \mathsf{cendcomplete}(H')$ and $S \in \mathcal{T}_A$ such that $H^c \sqsubseteq_{\mathsf{RT}} S$. Then H^c is obtained from H' by inserting some number of committed actions. Since $\mathcal{T}_A$ is closed under renaming action identifiers, we can assume that the identifiers of the committed actions do not occur in τ. Let τ_0 be a trace obtained from τ in the same way as H^c is obtained from H: by discarding abort-pending and live transactions and the same set of commit-pending transactions, while inserting the same committed actions, so that $\mathsf{history}(\tau^c) = H^c$. It is easy to see that $\tau^c \in [\![P]\!]_{\mathsf{noRB}}(s)$. In addition, $\tau^c|_{\neg\mathsf{trans}} = \tau|_{\neg\mathsf{trans}}$. Since $\tau^c \in [\![P]\!]_{\mathsf{noRB}}(s)$ and $H^c \sqsubseteq_{\mathsf{RT}} S$, by Lemma 6.2 there exists $\tau' \in [\![P]\!]_{\mathsf{noRB}}(s)$ such that $\mathsf{history}(\tau') = S$ and $\tau^c \approx \tau'$, which implies that $\tau'|_{\neg\mathsf{trans}} = \tau^c|_{\neg\mathsf{trans}} = \tau|_{\neg\mathsf{trans}}$ and τ' does not contain a fault. Hence, $\tau' \in [\![P, \mathcal{T}_A]\!]_{\mathsf{noRB}}(s)$ and $\tau \sim \tau'$.

Now assume that τ contains a `fault` action inside a live transaction. Let $\tau = \tau_1\psi\tau_2\chi$, where $\chi = (\_, t_0, \mathtt{fault})$ 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. Let $\tau_3 = \tau_1\psi\,(\tau_2|_{t_0})\,\chi$; then $\tau_3 \in [\![P, \mathcal{T}_C]\!]_{\mathsf{noRB}}(s)$. By our assumption, $\mathcal{T}_C \sqsubseteq_{\mathsf{STMS}} \mathcal{T}_A$. Then there exists $H^c_\psi \in \mathsf{cSTMSpast}(\mathsf{history}(\tau_3))$ and $S \in \mathcal{T}_A$ such that $H^c_\psi \sqsubseteq_{\mathsf{RT}} S$. By Lemma 6.8, for some τ_4 we have $\tau_4 \in [\![P]\!]_{\mathsf{noRB}}(s)$, $\mathsf{history}(\tau_4) = H^c_\psi$, and $\tau_4|_{t_0} = \tau_3|_{t_0} = \_\chi$. By Lemma 6.2, there exists $\tau_5 \in [\![P, \mathcal{T}_A]\!]_{\mathsf{noRB}}(s)$ such that $\tau_4 \approx \tau_5$ and hence $\tau_5 = \_\chi$ and $\tau \sim \tau_5$.

6.4 Proof of Theorem 5.16(i): Sufficiency of the TMS Relation

To prove Theorem 5.16(i), we cannot straightforwardly apply Lemma 6.2: Definition 5.8(i) matches only histories of committed transactions, but the histories of traces produced by the program P in Lemma 6.2 also contain aborted transactions. Fortunately, the following Lemma 6.10 allows us to add empty aborted transactions into the abstract history while preserving the real-time order of all actions. Furthermore, the following proposition shows that under the semantics with rollback, the set of traces produced by a program is closed under making all aborted transactions empty.

Proposition 6.9. $\forall\tau.\, \tau \in [\![P]\!]_{\mathsf{RB}}(s) \Rightarrow \tau|_{\neg\mathsf{abortact}} \in [\![P]\!]_{\mathsf{RB}}(s)$.

We call a history H_a an *immediate abort extension of* H if $H_a \in \mathsf{addoneab}(H)$ (see C7 in Figure 8) or there exists an immediate abort extension H' of H such that $H_a \in \mathsf{addoneab}(H')$. We denote the set of all immediate abort extensions of H by $\mathsf{addab}(H)$.

Lemma 6.10. *Let* $H, S \in \mathsf{WfHistory}$ *be such that* $H|_{\neg\mathsf{abortact}} = H$ *and* $H|_{\neg\mathsf{aborted}} \sqsubseteq_{\mathsf{RT}} S$. *There exists* $S' \in \mathsf{WfHistory}$ *such that* $S' \in \mathsf{addab}(S)$ *and* $H \sqsubseteq_{\mathsf{RT}} S'$.

Proof. 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

$$\begin{gathered} S_i \in \mathsf{WfHistory}; \quad |\mathsf{aborted}(S_i)| = i; \quad S_i \in \mathsf{addab}(S); \\ \{\psi \mid \psi \in H|_{\neg\mathsf{aborted}}\} \subseteq \{\psi \mid \psi \in S_i\} \subseteq \{\psi \mid \psi \in H\}; \\ \forall \psi_1, \psi_2 \in S_i.\, \psi_1 \prec_H \psi_2 \Rightarrow \psi_1 \prec_{S_i} \psi_2. \end{gathered} \tag{10}$$

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

For $i = 0$, we take $S_0 = S$, and all requirements in (10) hold vacuously. Assume that a history S_i satisfying (10) 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 $\psi_b = (\_, t, \mathsf{txbegin})$, $\psi_a = (\_, t, \mathsf{aborted})$, $\psi_b \notin S_i$, $H_2|_t = \varepsilon$ and

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

In other words, out of all aborted transactions in H that are not in S_i, $\psi_b\psi_a$ is the one with the earliest txbegin.

If H_1 does not contain a committed or an aborted action, we let $S_{i+1} = \psi_b\psi_a S_i$. Then $S_{i+1} \in \mathsf{WfHistory}$. We only need to show that for any $\psi' \in S_i$, we have $\psi' \prec_H \psi_b \Rightarrow \psi' \prec_{S_{i+1}} \psi_b$ and $\psi_a \prec_H \psi' \Rightarrow \psi_a \prec_{S_{i+1}} \psi'$. 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'.

The rest of the proof deals with the case when 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 $\psi' \prec_H \psi_b \Rightarrow \psi' \prec_{S_{i+1}} \psi_b$ and $\psi_a \prec_H \psi' \Rightarrow \psi_a \prec_{S_{i+1}} \psi'$.

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 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 $\psi_a \prec_{S_{i+1}} \psi'$.

Finally, it is easy to show that $H_1|_t = (S'\psi)|_t$, which implies that $S_{i+1} \in \mathsf{WfHistory}$. □

Proof of Theorem 5.16(1). The proof is similar to that of Theorem 5.15(i). Assume $\mathcal{T}_C \sqsubseteq_{\mathsf{TMS}} \mathcal{T}_A$. Consider $\tau \in [\![P, \mathcal{T}_C]\!]_{\mathsf{RB}}(s)$, and let $H = \mathsf{history}(\tau)$.

Assume first that τ does not contain a `fault` action inside a live transaction. Since $\mathcal{T}_C \sqsubseteq_{\mathsf{TMS}} \mathcal{T}_A$, there exist $H' \in \mathsf{rempending}(H|_{\neg\mathsf{live}})$, $H^c \in \mathsf{cendcomplete}(H')$, and $S \in \mathcal{T}_A$ such that $H^c|_{\neg\mathsf{aborted}} \sqsubseteq_{\mathsf{RT}} S$. Then H^c is obtained from $H'|_{\neg\mathsf{live}}$ by inserting some number of committed actions. Without loss of generality, we can assume that the identifiers of these actions do not occur in τ. Let τ_0 be a trace obtained from τ in the same way as H^c is obtained from H so that $\mathsf{history}(\tau^c) = H^c$. It is easy to see that $\tau^c \in [\![P]\!]_{\mathsf{RB}}(s)$. Additionally, $\tau^c|_{\neg\mathsf{trans}} = \tau|_{\neg\mathsf{trans}}$. Let $\tau^{na} = \tau^c|_{\neg\mathsf{abortact}}$. By Proposition 6.9, we get $\tau^{na} \in [\![P]\!]_{\mathsf{RB}}(s)$. Since $(H^c|_{\neg\mathsf{abortact}})|_{\neg\mathsf{aborted}} = H^c|_{\neg\mathsf{aborted}} \sqsubseteq_{\mathsf{RT}} S$, by Lemma 6.10, for some history S' we have $H^c|_{\neg\mathsf{abortact}} \sqsubseteq_{\mathsf{RT}} S'$ and $S' \in \mathsf{addab}(S)$. Since $S \in \mathcal{T}_A$ and $\mathcal{T}_A$ satisfies C7, we have $S' \in \mathcal{T}_A$. We have $\tau^{na} \in [\![P]\!]_{\mathsf{RB}}(s)$ and $\mathsf{history}(\tau^{na}) = H^c|_{\neg\mathsf{abortact}}$; hence, by Lemma 6.2, there exists a trace $\tau' \in [\![P]\!]_{\mathsf{RB}}(s)$ such that $\mathsf{history}(\tau') = S' \in \mathcal{T}_A$ and $\tau^{na} \approx \tau'$, which implies that

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

and τ' does not contain a fault. Hence, $\tau' \in [\![P, \mathcal{T}_A]\!]_{\mathsf{RB}}(s)$ and $\tau \sim \tau'$.

Now assume that τ contains a `fault` action inside a live transaction. Let $\tau = \tau_1\psi\tau_2\chi$, 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. Let $\tau_3 = \tau_1\psi\,(\tau_2|_{t_0})\,\chi$; then $\tau_3 \in [\![P, \mathcal{T}_C]\!]_{\mathsf{RB}}(s)$. By our assumption, $\mathcal{T}_C \sqsubseteq_{\mathsf{TMS}} \mathcal{T}_A$. Then there exists $H^c_\psi \in \mathsf{cTMSpast}(\mathsf{history}(\tau_3))$ and $S \in \mathcal{T}_A$ such that $H^c_\psi \sqsubseteq_{\mathsf{RT}} S$. By Definition 5.7, there exists a history $H'_\psi \in \mathsf{cSTMSpast}(\mathsf{history}(\tau_3))$ such that $H^c_\psi = H'_\psi|_{\neg\mathsf{aborted}}$. By Lemma 6.8, for some τ_4, we have $\tau_4 \in [\![P]\!]_{\mathsf{RB}}(s)$, $\mathsf{history}(\tau_4) = H'_\psi$, and $\tau_4|_{t_0} = \tau_3|_{t_0} = \_\chi$. By Proposition 6.9, $\tau_4|_{\neg\mathsf{abortact}} \in [\![P]\!]_{\mathsf{RB}}(s)$. Using Lemma 6.10, we get $S' \in \mathsf{WfHistory}$ such that $\mathsf{history}(\tau_4|_{\neg\mathsf{abortact}}) \sqsubseteq_{\mathsf{RT}} S'$ and $S' \in \mathsf{addab}(S)$. Since $S \in \mathcal{T}_A$ and $\mathcal{T}_A$ satisfies C7, we get $S' \in \mathcal{T}_A$. Hence, by Lemma 6.2, there exists $\tau_5 \in [\![P, \mathcal{T}_A]\!]_{\mathsf{RB}}(s)$ such that $\tau_4 \approx \tau_5$, and hence $\tau_5 = \_\chi$ and $\tau \sim \tau'$. □

7 NECESSITY PROOFS

7.1 Proof of Theorem 5.15(ii): Necessity of the STMS Relation

Theorem 5.15(ii) follows from Lemmas 7.1 and 7.2, stated and proved in the following. The former establishes the conditions in Definition 5.6(i), and the latter establishes the conditions in Definition 5.6(ii).

Lemma 7.1. *Let $\mathcal{T}_C$ and $\mathcal{T}_A$ be TMs such that $\mathcal{T}_C \preceq_{\mathsf{noRB}} \mathcal{T}_A$ and $\mathcal{T}_A$ satisfies C1 and C2. The following holds:*

$$\forall H \in \mathcal{T}_C.\ \exists H' \in \mathsf{rempending}(H|_{\neg\mathsf{live}}).\ \exists H^c \in \mathsf{cendcomplete}(H').\ \exists S \in \mathcal{T}_A.\ H^c \sqsubseteq_{\mathsf{RT}} S. \tag{11}$$

Proof. The proof works in several stages, and this structure is common to all lemmas in our proofs of necessity:

I. For every history $H \in \mathcal{T}_C$, we construct a particular program P_H. Threads in P_H perform the sequence of transactions specified by H, record the return values that they obtain from the TM, and monitor whether the real-time order between actions includes the one in H. Crucially, if the TM behavior in an execution of P_H deviates significantly from H, then the program does not perform certain nontransactional actions that it would perform otherwise.
II. As we show, the program P_H is such that for some state s and trace $\tau \in [\![P_H]\!]_{\mathsf{noRB}}(s)$ without a fault, we have $\mathsf{history}(\tau) = H$.
III. Since $H \in \mathcal{T}_C$ and $\mathcal{T}_C \preceq_{\mathsf{noRB}} \mathcal{T}_A$, we get that there exists a trace $\tau' \in [\![P_H, \mathcal{T}_A]\!]_{\mathsf{noRB}}(s)$ such that $\tau \sim \tau'$. Since τ does not contain a fault, Definition 4.1 implies $\tau'|_{\neg\mathsf{trans}} = \tau|_{\neg\mathsf{trans}}$. Let $S_1 = \mathsf{history}(\tau') \in \mathcal{T}_A$. From $\tau'|_{\neg\mathsf{trans}} = \tau|_{\neg\mathsf{trans}}$ and the structure of P_H, we infer a certain correspondence between the histories H and S_1: these histories cannot be too far apart.
IV. The correspondence between H and S_1 allows us to transform these into histories H^c and S required by (11), with the transformations on S_1 justified using the closure properties C1 and C2.

Stage I: Constructing P_H. Consider a history $H \in \mathcal{T}_C$. We define the program P_H in Figure 15 using the auxiliary notation in Figure 16. For reference, Figure 17 lists the variables used in P_H and their intended meaning. Given that the history H is well formed, we construct the command C^t executed by every thread t in P_H as a sequence of `atomic` blocks, corresponding to txbegin, txcommit, txabort, committed, and aborted actions in $H|_t$. These blocks perform the sequence of method invocations determined by call and ret actions in $H|_t$. In more detail, the command C^t is comprised of a sequence of commands C^t_i each constructed according to the i-th transaction of t in H.

$$
\begin{aligned}
P_H &= C^1 \parallel \ldots \parallel C^m \\
C^t &= C^t_1; C^t_2; \ldots; C^t_{k^t} \\
C^t_i &= \texttt{if}\ (g^1_{\mathsf{lasttx}(t,i,1)} \neq 1)\ \texttt{then loop};\\
&\quad \ldots \\
&\quad \texttt{if}\ (g^m_{\mathsf{lasttx}(t,i,m)} \neq 1)\ \texttt{then loop} \\
&\quad w^t_i := \texttt{atomic}\ \{ \\
&\qquad y^t_{i,1} := o^t_{i,1}.f^t_{i,1}(n^t_{i,1}); \\
&\qquad \texttt{if}\ (y^t_{i,1} \neq r^t_{i,1})\ \texttt{then loop}; \\
&\qquad \ldots; \\
&\qquad y^t_{i,q^t_i-1} := o^t_{i,q^t_i-1}.f^t_{i,q^t_i-1}(n^t_{i,q^t_i-1}); \\
&\qquad \texttt{if}\ (y^t_{i,q^t_i-1} \neq r^t_{i,q^t_i-1})\ \texttt{then loop}; \\
&\qquad a^t_i = 1; \\
&\qquad y^t_{i,q^t_i} := o^t_{i,q^t_i}.f^t_{i,q^t_i}(n^t_{i,q^t_i}); \\
&\qquad \texttt{if}\ (y^t_{i,q^t_i} \neq r^t_{i,q^t_i})\ \texttt{then loop}; \\
&\qquad b^t_i = 1; \\
&\qquad \mathit{lastCommand}^t_i \\
&\quad \} \\
&\quad \texttt{if}\ (w^t_i \neq c^t_i)\ \texttt{then loop}; \\
&\quad \texttt{if}\ (\neg \mathit{lastCheck}^t_i)\ \texttt{then loop}; \\
&\quad g^t_i := 1
\end{aligned}
$$

$$
\mathit{lastCommand}^t_i = \begin{cases} \texttt{loop}, & H^t_i \in \mathsf{live}(H) \cup \mathsf{midaborted}(H); \\ \texttt{abort}, & H^t_i \in \mathsf{selfaborted}(H) \cup \mathsf{abpending}(H); \\ \texttt{skip}, & \text{otherwise}; \end{cases}
$$

$$
\mathit{lastCheck}^t_i = \begin{cases} a^t_i \neq 1, & H^t_i \in \mathsf{midaborted}(H) \text{ and } |H^t_i| = 2; \\ a^t_i = 1 \text{ and } b^t_i \neq 1, & H^t_i \in \mathsf{midaborted}(H) \text{ and } |H^t_i| \neq 2; \\ b^t_i = 1, & \text{otherwise}. \end{cases}
$$

Fig. 15. The construction of P_H. For conciseness, we use an extension of the programming language with conditionals without an "else" clause. The `loop` command is a syntactic sugar for `while (true) do skip`.

The command C^t_H records the return value of the j-th method invocation in the i-th transaction by thread t in a dedicated variable $y^t_{i,j}$, local to t. The return status of the i-th transaction is recorded in a dedicated local variable w^t_i. The command checks the values of $y^t_{i,j}$ and w^t_i, and if there is a mismatch with $H|_t$, the command enters a nonterminating loop. We also perform some additional checks using the variables a^t_i and b^t_i, which we explain later on. If the i-th transaction of thread t in H is live or mid-aborted, then the command C^t_H executes a nonterminating loop before the end of the corresponding atomic block, to ensure that the transaction does not try to commit. If the transaction aborts itself explicitly, then so does the command C^t_H.

Shorthand	Description
u	An arbitrary integer which does not appear in H
m	The largest thread identifier occurring in H
k^t	The number of transactions started by thread t in H, i.e., the number of $(\_, t, \mathsf{txbegin})$ actions in H
H_i^t	$H\|_t$ is partitioned 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, i.e., $H_i^t(1) = (\_, t, \mathsf{txbegin})$ and H_i^t contains a single $\mathsf{txbegin}$ action.
c_i^t	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 i-th transaction is not completed, then $c_i^t = u$
q_i^t	The number of call actions of thread t in its i-th transaction, i.e., in H_i^t
$(\_, t, \mathsf{call}\ o_{i,j}^t.f_{i,j}^t(n_{i,j}^t))$	The j-th call action of thread t in its i-th transaction
$(\_, t, \mathsf{ret}(r_{i,j}^t)\ o_{i,j}^t.f_{i,j}^t)$	The j-th ret action of thread t in its i-th transaction; if there is no response in H to $(\_, t, \mathsf{call}\ o_{i,j}^t.f_{i,j}^t(n_{i,j}^t))$ or the response is an $\mathsf{aborted}$ action, then $r_{i,j}^t = u$
$\mathsf{lasttx}(t, i, t')$	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

Fig. 16. Auxiliary notation derived from the history H and used to construct P_H (see Figure 15).

Variable	Description
$y_{i,j}^t$	Records the return value of the j-th object method invocation in the i-th transaction of thread t
w_i^t	Records whether the i-th transaction of thread t committed or aborted
g_i^t	Signals that the i-th transaction of thread t completed: g_i^t is set to 1 only after t's i-th transaction commits or aborts
a_i^t	Determines whether the i-th transaction of thread t was mid-aborted prior to its last method invocation
b_i^t	Determines whether the i-th transaction of thread t was mid-aborted

Fig. 17. The variables used in P_H and their their intended meaning (see Figure 15).

To check whether an execution of P_H complies with the real-time order in H, we exploit the ability of threads to communicate via global variables outside transactions. For each transaction in H, we introduce a global variable g_i^t, which is initially 0 and is set to 1 by t right after its i-th transaction completes. We also add a dummy variable g_0^t for every thread t; we always execute the program from a state in which all g_0^t are initialized to 1. Before starting the i-th transaction of thread t, the command C_i^t checks whether all transactions preceding it in the real-time order in H have finished by reading the corresponding g-variables. If there is a mismatch with H, the command enters a nonterminating loop.

Stage II: Constructing s and $\tau \in [\![P_H]\!]_{\mathsf{noRB}}(s)$ such that $\mathsf{history}(\tau) = H$. We now construct a particular trace τ of P_H by first constructing a trace τ^t for every sequential command C_H^t and then interleaving the traces $\tau^1, \ldots, \tau^m$ in a particular way.

Let τ^t be the maximal trace from $\mathsf{Tr}(C^t)t$ such that $\mathsf{history}(\tau^t) = H|_t$ and τ^t does not include actions coming from a `loop` command. This trace τ^t exists because $\mathsf{Tr}(C^t)t$ contains traces for every possible parameter and return value of method invocations and `atomic` blocks in C_H^t (Figure 4). We now partition every τ^t into $|H|_t| + 1$ subsequences that we later interleave to create τ:

$$\tau^t = \tau_1^t \ldots \tau_{|H|_t|}^t \tau_{|H|_t|+1}^t.$$

Formally, for every $i = 1..|H|_t|$, there is exactly one interface action ψ_i^t in τ_i^t and the following holds:

$$\tau_i^t = \begin{cases} \psi_i^t, & \text{if } \psi_i^t \in \{(\_, t, \mathsf{OK}), (\_, t, \mathsf{ret}(\_)\ \_)\}; \\ \psi_i^t\_(\_, t, g_i^t := 1), & \text{if } \psi_i^t \in \{(\_, t, \mathsf{committed}), (\_, t, \mathsf{aborted})\}; \\ \_\psi_i^t, & \text{otherwise.} \end{cases} \tag{12}$$

Note that this defines τ_i^t, $i = 1..|H|_t|$, uniquely. Therefore, $\tau_{|H|_t|+1}^t$ is also defined uniquely as containing the rest of the actions in τ^t. Because each subsequence τ_i^t, except the last one, contains exactly one interface action, we can now construct the desired τ by interleaving the subsequences in the order induced by H:

$$\tau = \tau_{j_1}^{t^1} \ldots \tau_{j_{|H|}}^{t^{|H|}} \tau_{|H|_1|+1}^1 \ldots \tau_{|H|_m|+1}^m, \text{ where } H(i) = (\_, t^i, \_),\ j_i = |(H\downarrow_i)|_{t^i}|. \tag{13}$$

Then $\mathsf{history}(\tau) = H$. Since $\tau^t \in \mathsf{Tr}(C^t)t$, we have $\tau \in \mathsf{Tr}(P_H)$. Let s be the state where all local variables are set to u, and for all t we have $g_i^t = 0$ for $i \neq 0$ and $g_0^t = 1$. It is easy to check that $\mathsf{eval}_{\mathsf{noRB}}(s, \tau) \neq \emptyset$, and hence $\tau \in [\![P_H]\!]_{\mathsf{noRB}}(s)$. Since $H \in \mathcal{T}_C$, we furthermore get $\tau \in [\![P_H]\!]_{\mathsf{noRB}}(s, \mathcal{T}_C)$.

Stage III: Obtaining $\tau' \in [\![P_H, \mathcal{T}_A]\!]_{\mathsf{noRB}}(s)$ and analyzing the relationship between its history and H. By assumption, $\mathsf{history}(\tau) = H \in \mathcal{T}_C$. Since $\tau \in [\![P_H]\!]_{\mathsf{noRB}}(s, \mathcal{T}_C)$ and $\mathcal{T}_C \preceq_{\mathsf{noRB}} \mathcal{T}_A$, by Definition 4.1 there exists a trace $\tau' \in [\![P_H]\!]_{\mathsf{noRB}}(s, \mathcal{T}_A)$ such that $\tau'|_{\neg\mathsf{trans}} = \tau|_{\neg\mathsf{trans}}$ and $S_1 = \mathsf{history}(\tau') \in \mathcal{T}_A$.

Consider a thread t. Let T be the i-th transaction in $H|_t$ and T' be the i-th transaction in $S_1|_t$, which we call the transaction *matching* T. 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 of P_H and $\tau'|_{\neg\mathsf{trans}} = \tau|_{\neg\mathsf{trans}}$ ensure the following:

(i) If T is committed, then T' exists and $T \equiv T'$. This is due to the checks of w_i^t and $y_{i,\_}^t$ in P_H. These checks succeed in τ, so that the assignment to g_i^t is executed. Since $\tau'|_{\neg\mathsf{trans}} = \tau|_{\neg\mathsf{trans}}$, the assignment to g_i^t is executed in τ' as well, so the checks also succeed in τ'.

(ii) If T is end-aborted, then T' exists and $T \equiv T'$. This is due to the checks of w_i^t, $y_{i,\_}^t$, and b_i^t in P_H. In particular, since T is end-aborted, the check for $b_i^t = 1$ in $lastCheck_i^t$ succeeds in τ; hence, the check also has to succeed in τ', and so T' cannot be mid-aborted.

(iii) If T is mid-aborted, then T' exists and $T \equiv T'$. This is due to the checks of w_i^t, $y_{i,\_}^t$, b_i^t and a_i^t in P_H. In particular, if T is mid-aborted at its txbegin action (so that $|H_i^t| = 2$), then the check for $a_i^t \neq 1$ in $lastCheck_i^t$ ensures that T' is also mid-aborted at its txbegin action. If T is mid-aborted at its last method invocation (so that $|H_i^t| \neq 2$), then the check for $b_i^t \neq 1$ and $a_i^t = 1$ in $lastCheck_i^t$ ensures that T' behaves the same.

(iv) If T is self-aborted, then T' exists and $T \equiv T'$. This is due to the checks of w_i^t, $y_{i,\_}^t$, and b_i^t in P_H.

(v) If T is live, then the nonterminating loop before the end of the corresponding atomic block in P_H ensures that T' is live, mid-aborted, or does not exist.

Case	T in H	T' in S_1	Transformation on T	Transformation on T'
(i)	committed	committed	leave as is	leave as is
(ii)	end-aborted	end-aborted	leave as is	leave as is
(iii)	mid-aborted	mid-aborted	leave as is	leave as is
(iv)	live	live, mid-aborted, does not exist	remove	remove
(v)	commit-pending	committed	commit	leave as is
(v)	commit-pending	live, aborted, does not exist	remove	remove
(v)	commit-pending	commit-pending	leave as is	leave as is
(vi)	self-aborted	self-aborted	leave as is	leave as is
(vii)	abort-pending	live, aborted, abort-pending, does not exist	remove	remove

Fig. 18. Summary of transformations used to generate H^c from $H = \text{history}(\tau)$ and S from $S_1 = \text{history}(\tau')$ in Lemma 7.1. The "Case" column refers to a relationship established between T and T' in Stage III of the proof. The "commit" and "abort" transformations refer to adding a matching committed or aborted action at the end of H.

(vi) If T is commit-pending, then T' may have any status or may not exist at all. However, if T' is visible, then the checks of $y^t_{i,\_}$ ensure that the return values of method invocations inside T and T' match.

(vii) If T is abort-pending, then T' may have any status or may not exist at all.

Finally, consider another transaction T_1 in H and its matching transaction T'_1 in S_1. Then:

(viii) If $T \prec_H T_1$ and T'_1 exists, then T' exists and $T' \prec_{S_1} T'_1$. This is because in τ the value of g^t_i is set to 1 after the transaction corresponding to T completes, and g-variables are checked before the transaction corresponding to T_1 starts (see (12)).

Stage IV: Constructing the desired H^c and S. From the preceding analysis, it follows that for any thread t, the i-th transaction in $H|_t$, except possibly the last one, is the same as the i-th transaction in $S_1|_t$. We now construct H^c from H and S from S_1 by applying certain transformations that reconcile the differences between H and S_1. The resulting H^c and S are such that

$$\exists H'.\, H' \in \mathsf{rempending}(H|_{\neg\mathsf{live}}) \wedge H^c \in \mathsf{cendcomplete}(H') \wedge S \in \mathcal{T}_A \wedge H^c \sqsubseteq_{\mathsf{RT}} S.$$

We start by applying the transformations in Figure 18 to each transaction T in H and its matching transaction T' in S_1. This relies on the analysis of the relationship between T and T' performed in Stage III, and we reference the corresponding cases in the figure. Let H_2 and S_2 be the resulting histories obtained from H and S_1, respectively. Given the relationship between H and S_1 established in Stage III, it is easy to see that $\forall t.\, H_2|_t \equiv S_2|_t$. Since $S_1 \in \mathcal{T}_A$ and $\mathcal{T}_A$ satisfies closure property C1, we have $S_2 \in \mathcal{T}_A$. Let S_3 be the history obtained from S_2 by renaming action identifiers so that $\forall t.\, H_2|_t = S_3|_t$. Then by item (viii) presented previously, we furthermore have $H_2 \sqsubseteq_{\mathsf{RT}} S_3$. Since

$S_2 \in \mathcal{T}_A$ and $\mathcal{T}_A$ is closed under renaming action identifiers, we also get $S_3 \in \mathcal{T}_A$. Since $\mathcal{T}_A$ satisfies C2, there exists $S \in \mathsf{complete}(S_3) \cap \mathcal{T}_A$. We now construct H^c from H_2 by removing those commit-pending transactions in H_2 whose matching transactions in S_3 become aborted in S, and by adding the committed actions inserted into S_3 to obtain S at the end of the resulting history. This makes H^c complete, and it is easy to check that furthermore

$$\exists H'.\, H' \in \mathsf{rempending}(H|_{\neg\mathsf{live}}) \wedge H^c \in \mathsf{cendcomplete}(H').$$

Since $H_2 \sqsubseteq_{\mathsf{RT}} S_3$, we also have $H^c \sqsubseteq_{\mathsf{RT}} S$, as required. □

Lemma 7.2. *Let $\mathcal{T}_C$ and $\mathcal{T}_A$ be TMs such that $\mathcal{T}_C \preceq_{\mathsf{noRB}} \mathcal{T}_A$ and $\mathcal{T}_A$ satisfies C1 and C2. Let $H_1\psi H_2 \in \mathcal{T}_C$, where ψ is a response action that is not a* committed *or* aborted *action. Then $\exists H^c \in \mathsf{cSTMSpast}(H_1\psi).\ \exists S \in \mathcal{T}_A.\ H^c \sqsubseteq_{\mathsf{RT}} S$.*

Proof. The overall structure of the proof is similar to that of Lemma 7.1. Let $\psi = (\_, t_0, \_)$. Consider a history $H_1\psi H_2 \in \mathcal{T}_C$, where ψ is a response action that is not a committed or an aborted action. Let $H = H_1\psi$; since $\mathcal{T}_C$ is prefix-closed, we also have $H \in \mathcal{T}_C$.

Stage I: Constructing P_H. We define the program P_H as in Figure 15, but with the definition of *lastCommand* adjusted so that

$$lastCommand^{t_0}_{k^{t_0}} = \texttt{fault}. \tag{14}$$

Hence, thread t_0 finishes by executing a `fault` if it detects no mismatch with H in the TM behavior. This is motivated by the fact that faulting is the only observation that Definition 4.1 allows us to make about the behavior of the live transaction of ψ.

Stage II: Constructing s and $\tau \in [\![P_H]\!]_{\mathsf{noRB}}(s)$ such that $\mathsf{history}(\tau) = H$. We construct s and τ as in Lemma 7.1, but with (13) replaced by

$$\tau = \tau^{t^1}_{j_1} \ldots \tau^{t^{|H|}}_{j_{|H|}} \tau^{1}_{|H|_1|+1} \cdots \tau^{t_0-1}_{|H|_{t_0-1}|+1} \tau^{t_0+1}_{|H|_{t_0+1}|+1} \cdots \tau^{m}_{|H|_m|+1} \tau^{t_0}_{|H|_{t_0}|+1},$$

where $H(i) = (\_, t^i, \_)$, $j_i = |(H\downarrow_i)|_{t^i}|$. This ensures that `fault` is the last action of τ. Then we again have $\tau \in [\![P_H]\!]_{\mathsf{noRB}}(s, \mathcal{T}_C)$ and $\mathsf{history}(\tau) = H$.

Stage III: Obtaining $\tau' \in [\![P_H, \mathcal{T}_A]\!]_{\mathsf{noRB}}(s)$ and analyzing the relationship between its history and H. Since $\tau \in [\![P_H]\!]_{\mathsf{noRB}}(s, \mathcal{T}_C)$ ends with a fault and $\mathcal{T}_C \preceq_{\mathsf{noRB}} \mathcal{T}_A$, by Definition 4.1 there exists a trace $\tau' \in [\![P_H]\!]_{\mathsf{noRB}}(s, \mathcal{T}_A)$ that also ends with a fault, which also has to be by thread t_0. Let $S_1 = \mathsf{history}(\tau') \in \mathcal{T}_A$.

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

(i) If T is the last transaction in $H|_{t_0}$, then $T \equiv T'$. This is because τ' ends with a fault by t_0, and hence the checks of $y^{t_0}_{i,\_}$ in τ' succeed.

(ii) If T_1 is visible, then so is T. Indeed, if T is live or mid-aborted, then the nonterminating loop before the end of the corresponding atomic block ensures that T_1 cannot be visible. Furthermore, due to the checks of $y^{t}_{i,\_}$, in this case the return values of method invocations inside T and T_1 match.

(iii) If $T' \prec_H T$ and T_1 exists, then so does T'_1, and we have $T'_1 \prec_{S_1} T_1$ and the status of T' and T'_1 is the same. This is because before the transaction corresponding to T_1 starts in τ', there is a check that a g-variable is 1, and this variable is assigned to 1 only if the check of $w^{t'}_j$ succeeds. In fact, we also have the following stronger property.

(iv) If $T' \prec_H T$ and T_1 exists, then so does T'_1, and we have $T'_1 \prec_{S_1} T_1$ and $T' \equiv T'_1$. This is due to the checks of $y^{t'}_{j,\_}$, $a^{t'}_j$, $b^{t'}_j$, and $w^{t'}_j$.

Stage IV: Constructing the desired H^c and S. Let S_2 be the projection of S_1 that excludes all abort-pending and live transactions, except the one that came from the transaction in τ' with the fault. Since $S_1 \in \mathcal{T}_A$ and $\mathcal{T}_A$ satisfies closure property C1, $S_2 \in \mathcal{T}_A$. Since $S_2 \in \mathcal{T}_A$ and $\mathcal{T}_A$ satisfies C2, there exists a history $S_3^c \in \mathsf{complete}(S_2) \cap \mathcal{T}_A$.

Let H' be the subsequence of H consisting of

—the (live) transaction of ψ,
—any transaction such that its matching transaction in S_3^c exists and is committed, and
—any transaction that precedes any of the preceding transactions in the real-time order of H.

We now show that $H' \in \mathsf{STMSpast}(H)$. By the construction of H', we have

$$\forall T \in \mathsf{tx}(H').\, \forall T' \in \mathsf{tx}(H).\, T' \prec_H T \Rightarrow T' \in \mathsf{tx}(H'). \tag{15}$$

Next, we show

$$\mathsf{tx}(H') \subseteq \{\mathsf{txof}(\psi, H)\} \cup \mathsf{committed}(H) \cup \mathsf{aborted}(H) \cup \mathsf{compending}(H) \tag{16}$$

and

$$(\mathsf{midaborted}(H') \cup \mathsf{selfaborted}(H')) \cap \mathsf{maxtx}(H') = \emptyset. \tag{17}$$

Consider $T \in H'$. If $T = \mathsf{txof}(\psi, H)$, then T is live and hence is not self- or mid-aborted. Otherwise, by the construction of H', either T is not maximal in H' and hence is completed, or its matching transaction in S_3^c exists and is committed. Then its matching transaction in S_1 is visible. Hence, by item (ii), T is visible as well and, in particular, cannot be self- or mid-aborted. This establishes (16) and (17).

Since $\mathsf{txof}(\psi, H) \in H'$, (15) through (17) show that $H' \in \mathsf{STMSpast}(H)$. Let $H^c \in \mathsf{cendcomplete}(\mathsf{maxcom}(H'))$; then $H^c \in \mathsf{cSTMSpast}(H)$. Let S_3 be the subsequence of S_3^c consisting of those transactions that have matching transactions in H'. By the definition of H', this only removes aborted transactions from S_3. Since $S_3^c \in \mathcal{T}_A$ and $\mathcal{T}_A$ is closed under closure property C1, we get $S_3 \in \mathcal{T}_A$.

We now show that $H^c|_t \equiv S_3|_t$ for any t. Take an arbitrary transaction $T \in \mathsf{tx}(H^c|_t)$ and its matching transaction T' in H'. We consider several cases:

—If $T = \mathsf{txof}(\psi, H^c)$, then by item (i), T is equivalent to its matching transaction in S_1 and hence also that in S_3.
—Otherwise, if T' is maximal in H', then its matching transaction T'' in S_3^c exists and is committed. Then by item (ii), T' is visible and the return values of method invocations inside it match those in T''. By the definition of H^c, this implies $T \equiv T''$. Hence, T is also equivalent to its matching transaction in S_3.
—Finally, if T' is not maximal in H', then it is completed, and by item (iv), it is equivalent to its matching transaction in S_3^c. Then by the definition of H^c, T is equivalent to its matching transaction in S_3^c and hence also that in S_3.

We have thus established $\forall t.\, H^c|_t \equiv S_3|_t$. Let S be the history obtained from S_3 by renaming action identifiers such that we have $\forall t.\, H^c|_t = S|_t$. Since $S_3 \in \mathcal{T}_A$ and $\mathcal{T}_A$ is closed under renaming action identifiers, we get $S \in \mathcal{T}_A$. From item (iii), it follows that the real-time order of H^c is preserved in S. This gives us $H^c \sqsubseteq_{\mathsf{RT}} S$, as required. □

7.2 Proof of Theorem 5.16(ii): Necessity of the TMS Relation

Theorem 5.16(ii) follows from Lemmas 7.3 and 7.4, stated and proved in the following. The former establishes the conditions in Definition 5.8(i), and the latter establishes the conditions in Definition 5.8(ii).

Lemma 7.3. *Let $\mathcal{T}_C$ and $\mathcal{T}_A$ be TMs such that $\mathcal{T}_C \preceq_{\mathsf{RB}} \mathcal{T}_A$ and $\mathcal{T}_A$ satisfies C1 and C2. The following holds:*

$$\forall H \in \mathcal{T}_C.\, \exists H' \in \mathsf{rempending}(H|_{\neg\mathsf{live}}).\, \exists H^c \in \mathsf{cendcomplete}(H').\, \exists S \in \mathcal{T}_A.\, H^c|_{\neg\mathsf{aborted}} \sqsubseteq_{\mathsf{RT}} S.$$

Proof. The proof is similar to that of Lemma 7.1. Stages I and II go through as before, except we omit the assignments to a_i^t and b_i^t and the *lastCheck* commands from P_H in Figure 15. The rationale is that here we are working under the semantics with rollback and thus cannot gain any information from the assignments to a_i^t and b_i^t inside atomic blocks.

Stage III is again the same as in Lemma 7.1, except we no longer establish items (ii), (iii), and (iv) in the analysis of the correspondence between a transaction T from τ and a matching transaction T' in τ': if the transaction T is aborted, then the check of w_i^t still ensures that so is T'; however, we can make no conclusions about the behavior inside T'. We now describe how Stage IV of Lemma 7.1 is adapted to construct H^c from H and S from S_1 such that

$$\exists H'.\, H' \in \mathsf{rempending}(H|_{\neg\mathsf{live}}) \wedge H^c \in \mathsf{cendcomplete}(H') \wedge S \in \mathcal{T}_A \wedge H^c|_{\neg\mathsf{aborted}} \sqsubseteq_{\mathsf{RT}} S.$$

We again start by applying the transformations in Figure 18 to each transaction T in H and its matching transaction T' in S_1. Let H_2 and S_2 be the resulting histories obtained from H and S_1. Given the analysis in Stage III, it is easy to check that $\forall t.\, (H_2|_{\neg\mathsf{aborted}})|_t \equiv (S_2|_{\neg\mathsf{aborted}})|_t$. Since $S_1 \in \mathcal{T}_A$ and $\mathcal{T}_A$ is closed under closure property C1, we have $S_2|_{\neg\mathsf{aborted}} \in \mathcal{T}_A$. Let S_3 be a well-formed history obtained from S_2 by renaming action identifiers so that $\forall t.\, (H_2|_{\neg\mathsf{aborted}})|_t = (S_3|_{\neg\mathsf{aborted}})|_t$. Then by item (viii) from the proof of Lemma 7.1, we furthermore have $H_2|_{\neg\mathsf{aborted}} \sqsubseteq_{\mathsf{RT}} S_3|_{\neg\mathsf{aborted}}$. Since $S_2|_{\neg\mathsf{aborted}} \in \mathcal{T}_A$ and $\mathcal{T}_A$ is closed under renaming action identifiers, we also get $S_3|_{\neg\mathsf{aborted}} \in \mathcal{T}_A$. Since $\mathcal{T}_A$ is closed under closure property C2, there exists $S_4 \in \mathsf{complete}(S_3|_{\neg\mathsf{aborted}}) \cap \mathcal{T}_A$, and we let $S = S_4|_{\neg\mathsf{aborted}}$. Since $\mathcal{T}_A$ is closed under closure property C1, we get $S \in \mathcal{T}_A$. Without loss of generality, we can assume that the identifiers of actions inserted into S_3 to obtain S_4 do not occur in H_2. We now construct H^c by removing those commit-pending transactions in H_2 whose matching transactions in S_3 become aborted in S_4 and by adding the committed actions inserted into S_3 to obtain S_4 at the end of the resulting history. This makes H^c complete, and it is easy to check that furthermore

$$\exists H'.\, H' \in \mathsf{rempending}(H|_{\neg\mathsf{live}}) \wedge H^c \in \mathsf{cendcomplete}(H') \wedge H^c|_{\neg\mathsf{aborted}} \sqsubseteq_{\mathsf{RT}} S. \qquad \square$$

Lemma 7.4. *Let $\mathcal{T}_C$ and $\mathcal{T}_A$ be TMs such that $\mathcal{T}_C \preceq_{\mathsf{RB}} \mathcal{T}_A$ and $\mathcal{T}_A$ satisfies C1 and C2. Let $H_1\psi H_2 \in \mathcal{T}_C$, where ψ is a response action that is not a* committed *or an* aborted *action. Then $\exists H^c \in \mathsf{cTMSpast}(H_1\psi).\, \exists S \in \mathcal{T}_A.\, H^c \sqsubseteq_{\mathsf{RT}} S$.*

Proof. The proof is virtually identical to that of Lemma 7.2. Consider a history $H_1\psi H_2 \in \mathcal{T}_C$, where ψ is a response action that is not a committed or an aborted action. Let $H = H_1\psi$; then $H \in \mathcal{T}_C$. Stages I and II go through as before, except we change P_H in Figure 15 by omitting the assignments to a_i^t and b_i^t and the *lastCheck* commands, and by adjusting the definition of *lastCommand* so that (14) holds. Stage III is again the same, except we no longer establish item (iv): under the semantics with rollback, we can make no conclusions about return values inside aborted transactions.

In Stage IV, we construct $H^c \in \mathsf{cSTMSpast}(H)$ and $S_3 \in \mathcal{T}_A$ as before, except now, due to the absence of item (iv), we only show that $\forall t.\, (H^c|_{\neg\mathsf{aborted}})|_t \equiv (S_3|_{\neg\mathsf{aborted}})|_t$.

Now let $H_0 = H^c|_{\neg\mathsf{aborted}}$ and $S_0 = S_3|_{\neg\mathsf{aborted}}$. Then $\forall t.\, H_0|_t \equiv S_0|_t$ and $H_0 \in \mathsf{cTMSpast}(H)$. Since $S_3 \in \mathcal{T}_A$ and $\mathcal{T}_A$ is closed under closure property C1, we get $S_0 \in \mathcal{T}_A$. Let S be a well-formed history obtained from S_0 by renaming action identifiers so that $\forall t.\, H_0|_t = S|_t$. Since $\mathcal{T}_A$ is closed under renaming action identifiers, we also have $S \in \mathcal{T}_A$. As in Lemma 7.2, we infer that the real-time order in H_0 is preserved between the corresponding transactions in S, which gives us $H_0 \sqsubseteq_{\mathsf{RT}} S$. $\square$

8 RELATED WORK

Previous work has studied TM consistency by

- —investigating the semantics of different programming languages with atomic blocks and the feasibility of their efficient implementation (Harris et al. 2005; Abadi et al. 2008; Moore and Grossman 2008) or
- —defining consistency conditions for TM (Guerraoui and Kapalka 2011, 2011; Doherty et al. 2013; Imbs and Raynal 2012; Attiya et al. 2013b) and proving that particular TM implementations validate them (Guerraoui and Kapalka 2011; Bieniusa and Thiemann 2011).

Thus, previous work has tended to address the issue from the perspective of either programming languages or TM implementations and has not tried to relate these two levels in a formal manner. An exception is the work by Harris et al. (2006), which proved that a specific TM implementation—Bartok-STM—validates a particular semantics of atomic blocks in a programming language.

This article tries to fill in the gap in existing studies by relating the semantics of a programming language with atomic blocks to that of a TM system implementing them. Our work is complementary to previous proofs that particular TM systems satisfy particular consistency conditions (Guerraoui and Kapalka 2011; Bieniusa and Thiemann 2011), as it lifts such results to the language level. Our work is also more general than that of Harris et al. (2006), as our results allow establishing observational refinement for any TM implementation satisfying a particular consistency condition. However, some of the work mentioned previously (Abadi et al. 2008; Moore and Grossman 2008) investigated advanced language interfaces that we do not consider, such as nested transactions and access to shared data both inside and outside transactions.

This work employs a well-known technique from the theory of programming languages—observational refinement (He et al. 1986, 1987)—to explore the most appropriate way to specify TM consistency. Observational refinement has previously been used to characterize correctness criteria for libraries of concurrent data structures. Filipovic et al. (2009) proved that in this setting, sequential consistency (Lamport 1979) is necessary and sufficient for observational refinement, and so is linearizability (Herlihy and Wing 1990) when client programs can interact via shared global variables. Gotsman and Yang (2011) adjusted linearizability to account for infinite computations and showed its sufficiency for observational refinement in the case when the client can observe the validity of liveness properties. Our work takes this approach from the simpler setting of concurrent libraries to the more elaborate setup of TM. It is our hope that in the future, we can generalize our results to infinite computations, along the lines of Gotsman and Yang (2011).

Even though prior work has not formally connected the notion of observational refinement and TM consistency conditions, these conditions sometimes came with informal explanations connecting them to similar notions. In particular, Doherty et al. (2013) discussed why TMS 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 Theorem 5.16(i)—is informal, as their work lacks a formal model for programs and their semantics. Most of it explains how Definition 5.8(i) ensures the correctness of committed transactions. The discussion of the most challenging case of live transactions—corresponding to Definition 5.8(ii) and our Lemma 6.8—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 5.8(ii) ensure this. This reasoning is very delicate, as indicated by our proof of Lemma 6.8, which carefully selects which actions to erase when transforming the trace. Moreover, Doherty et al. do not argue that TMS is the weakest condition possible, as we established by our necessity result for the semantics with rollback.

A previous version of this work (Attiya et al. 2013a) established a connection between the opacity relation (Section 5.3) and a notion of observational refinement under the semantics without rollback that we introduced in Definition 6.3 (Section 6.1). As we explained before, this notion of refinement takes into account aspects of program behavior that are unobservable to the program user in practice. For this reason, in the current work, we consider a weaker notion of observational refinement and the STMS consistency condition, which is generally weaker than the opacity relation. The characterization of TMS using observational refinement was developed in another precursor of the present work (Attiya et al. 2014).

This article considers three TM consistency conditions—TMS, STMS, and opacity—but there are more. VWC (Imbs and Raynal 2012) is weaker than opacity but incomparable to TMS. Like TMS, it 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. DU-opacity (Attiya et al. 2013b) andTMS2 (Doherty et al. 2013) are stronger than opacity. These conditions put additional restrictions on the justifying history: DU-opacity restricts the set of transactions that a transaction is allowed to read from, whereas TMS2 requires the justifying history to include transactions in their completion order in the original history. As follows from our results, these conditions are stronger than necessary for observational refinement.

Similarly to other works using observational refinement to study consistency conditions (Gotsman and Yang 2011; Filipovic et al. 2009), we formulate STMS, TMS, and the opacity relation so that they are not restricted to a particular abstract TM $\mathcal{T}_A$. This generality 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. This generality is aligned with the approach that Siek and Wojciechowski (2015) take for specifying *last-use opacity*, which allows early release (Herlihy et al. 2003; Marathe et al. 2005). Cast in our terminology, last-use opacity relates a concrete TM (implementation) to an abstract TM that is not atomic. However, some of the closure properties that we require of abstract TMs, such as C1, preclude early release; lifting this limitation is future work.

9 CONCLUSION

This article has presented an approach for evaluating TM consistency conditions from distributed computing theory using the notion of observational refinement from programming language theory. We introduced STMS, a new consistency condition, and proved that it is necessary and sufficient for observational refinement for a programming language where local variables modified by a transaction are not rolled back upon an abort. STMS is derived from TMS (Doherty et al. 2013) but, unlike the latter, requires the abstract history to include all previously aborted transactions. We further proved that TMS is necessary and sufficient for observational refinement for a programming language where local variables are rolled back upon an abort. Finally, we established that STMS is equivalent to opacity under certain assumptions on the TM, requiring a liveness property and the possibility of explicit aborts. Our results demonstrate how TM consistency requirements are subtly affected by features of the programming model. We believe that the approach to evaluating TM consistency conditions that we advocate will enable TM implementors and language designers to make better-informed decisions. It also reduces the effort of proving that a TM implements its programming language interface correctly, by only requiring its developer to show that it satisfies the corresponding consistency condition. We hope that in the future, our approach can be

scaled to settings more complex than the one we considered, including nested transactions (Moss and Hosking 2006; Ni et al. 2007) and nontransactional access to transactional data (Spear et al. 2007).

REFERENCES

Martín Abadi, Andrew Birrell, Tim Harris, and Michael Isard. 2008. Semantics of transactional memory and automatic mutual exclusion. In *Proceedings of POPL*. ACM, New York, NY, 63–74.

Hagit Attiya, Alexey Gotsman, Sandeep Hans, and Noam Rinetzky. 2013a. A programming language perspective on transactional memory consistency. In *Proceedings of PODC*. ACM, New York, NY, 309–318.

Hagit Attiya, Alexey Gotsman, Sandeep Hans, and Noam Rinetzky. 2014. Safety of live transactions in transactional memory: TMS is necessary and sufficient. In *Proceedings of DISC*. 376–390.

Hagit Attiya, Sandeep Hans, Petr Kuznetsov, and Srivatsan Ravi. 2013b. Safety of deferred update in transactional memory. In *Proceedings of ICDCS*. IEEE, Los Alamitos, CA, 601–610.

Annette Bieniusa and Peter Thiemann. 2011. Proving isolation properties for software transactional memory. In *Proceedings of ESOP*. 38–56.

Luke Dalessandro, Michael F. Spear, and Michael L. Scott. 2010. NOrec: Streamlining STM by abolishing ownership records. In *Proceedings of PPOPP*. ACM, New York, NY, 67–78.

David Dice, Ori Shalev, and Nir Shavit. 2006. Transactional locking II. In *Proceedings of DISC*. 194–208.

Simon Doherty, Lindsay Groves, Victor Luchangco, and Mark Moir. 2013. Towards formally specifying and verifying transactional memory. *Formal Aspects of Computing* 25, 5, 769–799.

Ivana Filipovic, Peter W. O'Hearn, Noam Rinetzky, and Hongseok Yang. 2009. Abstraction for concurrent objects. In *Proceedings of ESOP*. 252–266.

Alexey Gotsman and Hongseok Yang. 2011. Liveness-preserving atomicity abstraction. In *Proceedings of ICALP*. 453–465.

Rachid Guerraoui and Michal Kapalka. 2008. On the correctness of transactional memory. In *Proceedings of PPOPP*. ACM, New York, NY 175–184.

Rachid Guerraoui and Michal Kapalka. 2011. *Principles of Transactional Memory*. Morgan & Claypool, San Rafael, CA.

T. Harris, J. Larus, and R. Rajwar. 2010. *Transactional Memory*. Morgan & Claypool, San Rafael, CA.

Tim Harris, Simon Marlow, Simon L. Peyton Jones, and Maurice Herlihy. 2005. Composable memory transactions. In *Proceedings of PPOPP*. ACM, New York, NY, 48–60.

Tim Harris, Mark Plesko, Avraham Shinnar, and David Tarditi. 2006. Optimizing memory transactions. In *Proceedings of PLDI*. ACM, New York, NY, 14–25.

J. He, C. Hoare, and J. Sanders. 1986. Data refinement refined. In *Proceedings of ESOP*. 187–196.

J. He, C. Hoare, and J. Sanders. 1987. Prespecification in data refinement. *Information Processing Letters* 25, 2, 71–76.

Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer III. 2003. Software transactional memory for dynamic-sized data structures. In *Proceedings of PODC*. ACM, New York, NY, 92–101.

Maurice Herlihy and J. Eliot B. Moss. 1993. Transactional memory: Architectural support for lock-free data structures. *ACM SIGARCH Computer Architecture News* 21, 2, 289–300.

Maurice Herlihy and Nir Shavit. 2008. *The Art of Multiprocessor Programming*. Morgan Kaufmann.

Maurice Herlihy and Jeannette M. Wing. 1990. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems* 12, 3, 463–492.

Damien Imbs and Michel Raynal. 2012. Virtual world consistency: A condition for STM systems (with a versatile protocol with invisible read operations). *Theoretical Computer Science* 444, 113–127.

Leslie Lamport. 1979. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers* 28, 9, 690–691.

Virendra J. Marathe, William N. Scherer III, and Michael L. Scott. 2005. Adaptive software transactional memory. In *Proceedings of DISC*. 354–368.

Katherine F. Moore and Dan Grossman. 2008. High-level small-step operational semantics for transactions. In *Proceedings of POPL*. ACM, New York, NY, 51–62.

J. Eliot B. Moss and Antony L. Hosking. 2006. Nested transactional memory: Model and architecture sketches. *Science of Computer Programming* 63, 2, 186–201.

Yang Ni, Vijay S. Menon, Ali-Reza Adl-Tabatabai, Antony L. Hosking, Richard L. Hudson, J. Eliot B. Moss, Bratin Saha, and Tatiana Shpeisman. 2007. Open nesting in software transactional memory. In *Proceedings of PPOPP*. 68–78.

Christos H. Papadimitriou. 1979. The serializability of concurrent database updates. *Journal of the ACM* 26, 4, 631–653.

Torvald Riegel, Pascal Felber, and Christof Fetzer. 2006. A lazy snapshot algorithm with eager validation. In *Proceedings of DISC*. 284–298.

Scala STM Expert Group. 2012. Scala STM Quick Start Guide. Retrieved October 29, 2017, from https://nbronson.github.io/scala-stm/quick_start.html.

Konrad Siek and Pawel T. Wojciechowski. 2015. Last-use opacity: A strong safety property for transactional memory with early release support. arXiv:1506.06275. http://arxiv.org/abs/1506.06275.

Michael F. Spear, Virendra J. Marathe, Luke Dalessandro, and Michael L. Scott. 2007. Privatization techniques for software transactional memory. In *Proceedings of PODC*. 338–339.

M. F. Spear, M. Silverman, L. Dalessandro, M. M. Michael, and M. L. Scott. 2008. Implementing and exploiting inevitability in software transactional memory. In *Proceedings of ICPP*. IEEE, Los Alamitos, CA, 59–66.

Adam Welc, Bratin Saha, and Ali-Reza Adl-Tabatabai. 2008. Irrevocable transactions and their applications. In *Proceedings of SPAA*. ACM, New York, NY, 285–296.

Received June 2016; revised July 2017; accepted August 2017