Safe Privatization in Transactional Memory

Artem Khyzha
IMDEA Software Institute
Universidad Politécnica de Madrid

Alexey Gotsman
IMDEA Software Institute

Hagit Attiya
Technion

Noam Rinetzky
Tel-Aviv University

Abstract

Transactional memory (TM) facilitates the development of concurrent applications by letting the programmer designate certain code blocks as atomic. Programmers using a TM often would like to access the same data both inside and outside transactions, e.g., to improve performance or to support legacy code. In this case, programmers would ideally like the TM to guarantee strong atomicity, where transactions can be viewed as executing atomically also with respect to non-transactional accesses. Since guaranteeing strong atomicity for arbitrary programs is prohibitively expensive, researchers have suggested guaranteeing it only for certain data-race free (DRF) programs, particularly those that follow the privatization idiom: from some point on, threads agree that a given object can be accessed non-transactionally. Supporting privatization safely in a TM is nontrivial, because this often requires correctly inserting transactional fences, which wait until all active transactions complete.

Unfortunately, there is currently no consensus on a single definition of transactional DRF, in particular, because no existing notion of DRF takes into account transactional fences. In this paper we propose such a notion and prove that, if a TM satisfies a certain condition generalizing opacity and a program using it is DRF assuming strong atomicity, then the program indeed has strongly atomic semantics. We show that our DRF notion allows the programmer to use privatization idioms. We also propose a method for proving our generalization of opacity and apply it to the TL2 TM.

1 Introduction

Transactional memory (TM) facilitates the development of concurrent applications by letting the programmer designate certain code blocks as atomic [22]. 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. This guarantee can be formalized as observational refinement [7]: every behavior a user can observe of a program using a TM implementation can also be observed when the program uses an abstract TM that executes each block atomically. A TM can be implemented in hardware [23, 28], software [39] or a combination of both [11, 26].

Often programmers using a TM would like to access the same data both inside and outside transactions. This may be desirable to avoid performance overheads of transactional accesses, to support legacy code, or for explicit memory deallocation. One typical pattern is privatization [41], illustrated in Figure 1(a). There the atomic blocks return a value signifying whether the transaction committed or aborted. In the program, an object \( x \) is guarded by a flag \( x_{\text{is\_private}} \), showing whether the object should be accessed transactionally (false) or non-transactionally (true). The left-hand-side thread first tries to set the flag inside transaction \( T_1 \), thereby privatizing \( x \). If successful, it then accesses \( x \) non-transactionally. A concurrent transaction \( T_2 \) in the right-hand-side thread checks the flag \( x_{\text{is\_private}} \) prior to accessing \( x \), to avoid simultaneous transactional and non-transactional access to the object. We expect the postcondition shown to hold: if privatization is successful, at the end of the program \( x \) should store 1, not 42. The opposite idiom is publication, illustrated in Figure 2. The left-hand-side thread writes to \( x \) non-transactionally and then clears the flag \( x_{\text{is\_private}} \) inside transaction \( T_1 \), thereby publishing...
x. The right-hand-side thread tests the flag inside transaction $T_2$, and if it is cleared, reads $x$. Again, we expect the postcondition to hold: if the right-hand-side thread sees the write to the flag, it should also see the write to $x$. The two idioms can be combined: the programmer may privatize an object, then access it non transactionally, and then publish it back for transactional access.

Ideally, programmers mixing transactional and non-transactional accesses to objects would like the TM to guarantee strong atomicity [21], where transactions can be viewed as executing atomically also with respect to non-transactional accesses, i.e., without interleaving with them. This is equivalent to considering every non-transactional access as a single-instruction transaction. For example, the program in Figure 3 under strongly atomic semantics can only produce executions where each of the non-transactional accesses $v_1$ and $v_2$ executes either before or after the transaction $T$, so that the postcondition in Figure 3 always holds.

Unfortunately, providing strong atomicity in software requires instrumenting non-transactional accesses with additional instructions for maintaining TM metadata. This undermines scalability and makes it difficult to reuse legacy code. Since most existing TMs are either software-based or rely on a software fall-back, they do not perform such instrumentation and, hence, provide weaker atomicity guarantees. For example, they may allow the program in Figure 3 to execute non-transactional accesses $v_1$ and $v_2$ between transactional writes to $x$ and $y$ and, thus, observe an intermediate state of the transaction, e.g., $x = 1$ and $y = 0$, which violates the postcondition in Figure 3.

Researchers have suggested resolving the tension between strong TM semantics and performance by taking inspiration from non-transactional shared-memory models, which are subject to the same problem: optimizations performed by processors and compilers weaken the guarantee of sequential consistency [27] ideal for this setting. The compromise taken is to guarantee sequential consistency for certain data-race free (DRF) programs, which do not access the same data concurrently without synchronization [6]. Racy programs either are allowed to produce non-sequentially-consistent behaviors [30], or are declared faulty and thus having no semantics at all [2]. DRF thus establishes a contract between the programmer and the run-time system, which can be formalized by the so-called Fundamental Property: if a program is DRF assuming the strong semantics (such as sequential consistency), then the program does have the strong semantics. The crucial feature of this property is that DRF is checked by considering only executions under the strong semantics, which relieves the programmer from having to reason about the weaker semantics of unrestricted programs. The DRF contract has formed the basis of the memory models of both Java [30] and C/C++ [2].

Applying the above approach to TM, strong atomicity would be guaranteed only for programs that do not have an analog of data races in this setting—informally, concurrent transactional and non-transactional accesses to the same data [4, 5, 8, 9, 34, 41]. For example, we do not want to guarantee strong atomicity for the program in Figure 3, which has such concurrent accesses to $x$ and $y$. On the other hand, the programs in Figure 1 and Figure 2 should be guaranteed strong atomicity, since at any point of time, an object is accessed either only transactionally or only non-transactionally. Unfortunately, whereas the DRF contract in non-transactional memory models has been worked out in detail, the situation in transactional models remains unsettled. There is currently no consensus on a single definition of transactional DRF: there are multiple competing proposals [4, 8, 9, 24, 29], which often come without a formal justification similar to the Fundamental Property of non-transactional memory models.
This paper makes a step towards a definition of transactional DRF on a par with solutions in non-transactional memory models. A key technical challenge we tackle is that many TM implementations, when used out-of-the-box, do not guarantee strong atomicity for seemingly well-behaved programs using privatization, such as the ones in Figure 1 [12, 15, 31, 38]. For example, such TMs may invalidate the postcondition of the program in Figure 1(a) due to the delayed commit problem [41]. In more detail, many software TM implementations execute transactions optimistically, buffering their writes, and flush them to memory only on commit. In this case, it is possible for the transaction $T_1$ to privatize $x$ and for $v$ to modify it after $T_2$ started committing, but before its write to $x$ reached the memory, so that $T_2$’s write subsequently overwrites $v$’s write and violates the postcondition. TMs that make transactional updates in-place and undo them on abort are subject to a similar problem. In Figure 1(b) we give another privatization example that is prone to a different problem—the doomed transaction problem [41]. A TM may execute $T_2$’s read from $x_{is\_private}$, and then $T_1$ and $v$. Because $T_1$ modifies $x_{is\_private}$, at this point $T_2$ is “doomed”, i.e., guaranteed to abort if it finishes executing. But if the non-transactional write $v$ is uninstrumented and ignores the metadata the TM maintains to ensure the consistency of reads, $T_2$ will read the value written to $x$ by $v$ and enter an infinite loop. This would never happen under strong atomicity, where $T_1$ and $v$ may not execute while $T_2$ is running.

A possible solution to the above problems is for the compiler or the programmer to insert special transactional fences [41]. These have semantics similar to read-copy-update (RCU) [32]: a fence blocks until all the transactions that were active when it was invoked complete, by either committing or aborting. For example, assume we insert a fence between the transaction $T_1$ and the non-transactional access $v$ in Figure 1(a). Then the delayed commit problem does not arise: if $T_2$ enters the if body and writes to $x$, then it must begin before the fence does; thus the fence will wait until $T_2$ completes and flushes its write to memory, so that $T_2$ cannot incorrectly overwrite $v$. Analogously, a fence between $T_1$ and $v$ in Figure 1(b) ensures that the doomed transaction problem does not arise: if $T_2$ reaches the while loop, then $v$ cannot execute before $T_2$ finishes, and thus the while loop immediately terminates.

Unfortunately, inserting transactional fences conservatively after every transaction, even when not required, undermines scalability. For example, Yoo et al. [43] showed that unnecessarily fencing a selection of transactional benchmarks leads to overheads of 32% on average and 107% in the worst case, the latter on one of the STAMP benchmarks [33]. For this reason, researchers have suggested placing transactional fences selectively, e.g., according to programmer annotations [43]. However, omitting fences without violating strong atomicity is nontrivial: for example, for several years the TM implementation in the GCC compiler had a buggy placement of transactional fences that omitted them after read-only transactions; this has recently been shown to violate strong atomicity [44]. To make sure such bugs are not habitual, we need a notion of transactional DRF that would take into account selective fence placements.

In this paper we propose just such a notion and formalize its Fundamental Property using observational refinement: if a program is DRF under strong atomicity (formalized as transactional sequential consistency [8, 9]), then all its executions are observationally equivalent to strongly atomic ones. We furthermore prove that the Fundamental Property holds under a certain condition on the TM, generalizing opacity [19, 20], which we call strong opacity. Thus, similarly to non-transactional memory models, the programmer writing code that has no data races according to our notion never needs to reason about weakly atomic semantics.

Our results thus establish a contract between client programs and TM implementations sufficient for strong atomicity. Of course, for this contract to be useful, it should not overconstrain either of its participants: programmers should be able to use the typical programming idioms, and common TM implementations should satisfy strong opacity that we require. In this paper we argue that this is indeed the case.

On the client side, our DRF notion allows the programmer to use privatization and publication idioms—programs in Figure 2 and in Figure 1 with a fence between $T_1$ and $v$ are considered data-race free and thus guaranteed strong atomicity. We hence view privatization and publication idioms as just particular ways of ensuring data-race freedom.

To justify appropriateness of our requirements on TM systems, we develop a method for proving that a TM satisfies strong opacity for DRF programs. Our method is modular: it requires only a minimal adjustment to a proof of the usual opacity of the TM assuming no mixed transactional/non-transactional accesses. We demonstrate the effectiveness of the method by applying it to prove the strong opacity of a realistic TM, TL2 [12], enhanced with transactional fences implemented using RCU. Our proof shows that this TM will indeed guarantee strong atomicity to programs satisfying our notion of DRF.

Thus, this paper makes the first proposal of transactional DRF that considers a flexible programming model (with transactional fences) and comes with a formal justification of its appropriateness—the Fundamental Property and the notion of TM correctness required for it to hold.

## 2 Programming Language

We now introduce a simple programming language with mixed transactional/non-transactional accesses, for which we formalize our results. We also define the semantics of the language when using a given TM implementation. As a special case of this semantics, we get the notion of strong...
atomicty [21] (also called transactional sequential consistency [8]): this is obtained by instantiating our semantics with a special idealized atomic TM where the execution of transactions does not interleave with that of other transactions or with non-transactional accesses.

2.1 Programming Language Syntax

A program \( P = C_1 || \ldots || C_N \) in our language is a parallel composition of commands \( C_i \) executed by different threads \( t \in \text{ThreadID} = \{1, \ldots, N\} \). Every thread \( t \in \text{ThreadID} \) has a set of local variables \( l \in \text{LVar}_t \), which only it can access; for simplicity, we assume that these are integer-valued. Threads have access to a transactional memory (TM), which manages a fixed collection of shared register objects \( x \in \text{Reg} \). The syntax of commands \( C \in \text{Com} \) is as follows:

\[
C = c \mid C \mid \text{if} (b) \text{then} C \text{else} C \mid \text{while} (b) \text{do} C
\]

where \( b \) and \( e \) denote Boolean, respectively, integer expressions over local variables and constants. The language includes primitive commands \( c \in \text{PComm} \), which operate on local variables, and standard control-flow constructs.

An atomic block \( l := \text{atomic}[C] \) executes \( C \) as a transaction, which the TM can commit or abort. The system’s decision is returned in the local variable \( l \), which gets assigned a distinguished value committed or aborted. We do not allow programs to abort a transaction explicitly, and forbid nested atomic blocks and, hence, nested transactions.

Commands can invoke two methods on a shared register \( x \): \( x.\text{read()} \) returns the current value of \( x \), and \( x.\text{write}(e) \) sets it to \( e \). Threads may call these methods both inside and outside atomic blocks. We refer to the former as a transactional accesses and to the latter as a non-transactional accesses. To make our presentation more approachable, following [20] we assume that each write in a single program execution writes a distinct value. Finally, the language includes a transactional fence command \( \text{fence} \), which acts as explained in §1. It may only be used outside transactions.

The simplicity of the above language allows us to clearly explain our contributions. We leave handling advanced features, such as nested transactions [35, 36] and nested synchronization [40] as future work.

2.2 A Trace-based Model of Computations

To define the semantics of our programming language, we need a formal model for program computations. To this end, we introduce traces—certain finite sequences of actions, each describing a single computation step (for simplicity, in this paper we consider only finite computations). Let \( \text{ActionId} \) be a set of action identifiers. Actions are of two kinds. A primitive action denotes the execution of a primitive command and is of the form \( (a, t, c) \), where \( a \in \text{ActionId}, t \in \text{ThreadID} \) and \( c \in \text{PComm} \). A TM interface action has one of the forms shown in Figure 4. We use \( \alpha \) to range over actions.

<table>
<thead>
<tr>
<th>Request actions</th>
<th>Matching response actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>((a, t, \text{txbegin}))</td>
<td>((a, t, \text{ok}) \mid (a, t, \text{aborted}))</td>
</tr>
<tr>
<td>((a, t, \text{txcommit}))</td>
<td>((a, t, \text{committed}) \mid (a, t, \text{aborted}))</td>
</tr>
<tr>
<td>((a, t, \text{write}(x, v)))</td>
<td>((a, t, \text{ret}(\bot)) \mid (a, t, \text{aborted}))</td>
</tr>
<tr>
<td>((a, t, \text{read}(x)))</td>
<td>((a, t, \text{ret}(v)) \mid (a, t, \text{aborted}))</td>
</tr>
<tr>
<td>((a, t, \text{fbegin}))</td>
<td>((a, t, \text{fend}))</td>
</tr>
</tbody>
</table>

Figure 4. TM interface actions. Here \( a \in \text{ActionId}, t \in \text{ThreadID}, x \in \text{Reg}, \text{and} v \in \mathbb{Z} \).

TM interface actions denote the control flow of a thread \( t \) crossing the boundary between the program and the TM: request actions correspond to the control being transferred from the former to the latter, and response actions, the other way around. A txbegin action is generated upon entering an atomic block, and a txcommit action when a transaction tries to commit upon exiting an atomic block. The request actions \text{write}(x, v) \text{and} \text{read}(x) \text{denote invocations of the write, respectively, read methods of register} x; \text{a write action is annotated with the value} v \text{written. The response actions} \text{ret}(\bot) \text{and} \text{ret}(v) \text{denote the return from invocations of write, respectively, read methods of a register; the latter is annotated with the value} v \text{read. For reasons explained below, we consider non-transactional accesses to registers as calling into the TM, and hence use the same actions for them as for transactional accesses. The TM may abort a transaction (but not a non-transactional access) at any point when it is in control; this is recorded by an aborted response action. The actions fbegin and fend denote the beginning, respectively, the end of the execution of a fence command. In the following _ denotes an irrelevant expression.}

Definition 2.1. A trace \( \tau \) is a finite sequence of actions satisfying certain well-formedness conditions (stated informally due to space constraints; see [25, §A]):

- every action in \( \tau \) has a unique identifier;
- commands in actions executed by a thread \( t \) do not access local variables of other threads \( t' \neq t \);
- every \text{write} operation writes a unique value distinct from \( v_{\text{init}} \) (the initial value of each register);
- for every thread \( t \), the projection \( \tau^t \) of \( \tau \) onto the actions by \( t \) cannot contain a request action immediately followed by a primitive action;
- request and response actions are properly matched;
- actions denoting the beginning and the end of transactions are properly matched;
- non-transactional accesses execute atomically: if \( \tau = \tau_1 \alpha \tau_2 \), where \( \alpha \) is a read or a write request action by thread \( t \), and all the transactions of \( t \) in \( \tau_1 \) completed, then \( \tau_2 \) begins with a response to \( \alpha \);
- non-transactional accesses never abort;
- fence actions may not occur inside transactions; and
• fence blocks until all active transactions complete: if \( \tau = \tau_1 (., t, \text{txbegin}) \tau_2 (., t', \text{fbegin}) \tau_3 (., t', \text{fend}) \tau_4 \) then either \( \tau_2 \) or \( \tau_3 \) contains an action of the form \((., t, \text{committed})\) or \((., t, \text{aborted})\).

We denote the set of traces by \( \text{Trace} \) and the set of actions in a trace \( \tau \) by \( \text{act}(\tau) \). For a trace \( \tau = \tau_0 . . . \), where \( \tau_0 \) is also a trace, we say that \( \tau_0 \) is a prefix of \( \tau \).

A transaction \( T \) is a nonempty trace such that it contains actions by the same thread, begins with a \text{txbegin} \text{action} and only its last action can be a committed or an aborted action. A transaction \( T \) is: committed if it ends with a committed action, aborted if it ends with aborted, commit-pending if it ends with \text{txcommit}, and live, in all other cases. A transaction \( T \) is in a trace \( \tau \) if \( T \) is a subsequence of \( \tau \) and no longer transaction is. We let \( \text{txn}(\tau) \) be the set of transactions in \( \tau \).

We refer to TM interface actions in a trace outside of a transaction as non-transactional actions. We call a matching request/response pair of a read or a write a non-transactional access. We denote by \( \text{nontxn}(\tau) \) the set of non-transactional accesses in \( \tau \) and range over them by \( v \).

A history is a trace containing only TM interface actions; we use \( H, S \) to range over histories. Since histories fully capture the possible interactions between a TM and a client program, we often confine the notion of a TM and the set of histories it produces. Hence, a transactional memory \( \mathcal{H} \) is a set of histories that is prefix-closed and closed under renaming action identifiers. Note that histories include actions corresponding to non-transactional accesses, even though these may not be directly managed by the TM implementation. This is needed to account for changes to registers performed by such actions when defining the TM semantics: e.g., in the case when a register is privatized, modified non-transactionally and then published back for transactional access. Of course, a well-formed TM semantics should not impose restrictions on the placement of non-transactional actions, since these are under the control of the program.

2.3 Programming Language Semantics

The semantics of the programming language is the set of traces that computations of programs produce. Due to space constraints, we defer its formal definition to [25, §A] and describe only its high-level structure. A state of a program \( P \) is \( C_1 \parallel \ldots \parallel C_N \) records the values of all its variables: \( s \in \text{State} = (\{\varnothing \}^N \text{Var}_1) \rightarrow \mathbb{Z} \). The semantics of a program \( P \) is given by the set of traces \( [P, \mathcal{H}](s) \subseteq \text{Trace} \) it produces when executed with a TM \( \mathcal{H} \) from an initial state \( s \). To define this set, we first define the set of traces \( [P](s) \subseteq \text{Trace} \) that a program can produce when executed from \( s \) with the behavior of the TM unrestricted, i.e., considering all possible values the TM can return on reads and allowing transactions to commit or abort arbitrarily. This definition follows the intuitive semantics of our programming language. We then restrict \( [P](s) \) to the set of traces produced by \( P \) when executed with \( \mathcal{H} \) by selecting those traces that interact with the TM in a way consistent with \( \mathcal{H} \): \( [P, \mathcal{H}](s) = \{ \tau \mid \tau \in [P](s) \wedge \text{history}(\tau) \in \mathcal{H} \} \), where \( \text{history}(\cdot) \) projects to TM interface actions.

2.4 Strong Atomicity

We now define an idealized atomic TM \( \mathcal{H}_{\text{atomic}} \) where the execution of transactions does not interleave with that of other transactions or with non-transactional accesses. By instantiating the semantics of §2.3 with this TM, we formalize the strongly atomic semantics [21] (transactional sequential consistency [8, 9]). \( \mathcal{H}_{\text{atomic}} \) contains only histories that are non-interleaved, i.e., where actions by one transaction do not overlap with the actions of another transaction or of non-transactional accesses. Note that by definition actions pertaining to different non-transactional accesses cannot interleave. Note also that transactions in a non-interleaved history do not have to be complete. For example,

\[
H_0 = (., t_1, \text{txbegin})(., t_1, \text{ok})(., t_1, \text{write}(x, 1))(., t_1, \text{ret}(\bot)) (., t_1, \text{txcommit})(., t_2, \text{txbegin})(., t_2, \text{ok})(., t_2, \text{write}(x, 2))(., t_3, \text{read}(x))(., t_3, \text{ret}(1))
\]

is non-interleaved. We have to allow such histories in \( \mathcal{H}_{\text{atomic}} \), because they may be produced by programs in our language, e.g., due to a non-terminating loop inside an atomic block.

We define \( \mathcal{H}_{\text{atomic}} \) in such a way that the changes made by a live or aborted 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 [20]. To account for this, when defining \( \mathcal{H}_{\text{atomic}} \) we consider every possible completion of each commit-pending transaction in a history to either committed or an aborted one. Formally, we say that a history \( H^c \) is a completion of a non-interleaved history \( H \) if: (i) \( H^c \) is non-interleaved; (ii) \( H^c \) has no commit-pending transactions; (iii) \( H \) is a subsequence of \( H^c \); and (iv) any action in \( H^c \) which is not in \( H \) is either a committed or an aborted action. For example, we can obtain a completion of history \( H_0 \) above by inserting \((., t_1, \text{committed})\) after \((., t_1, \text{txcommit})\).

We define \( \mathcal{H}_{\text{atomic}} \) as the set of all non-interleaved histories \( H \) that have a completion \( H^c \) where every response action of a read \( x \) returns the value \( v \) in the last preceding write \( x, v \) action that is not located in an aborted or live transaction different from the one of the read; if there is no such write, the read should return the initial value \( v_{\text{init}} \). For example, \( H_0 \in \mathcal{H}_{\text{atomic}} \). Hence, \( \mathcal{H}_{\text{atomic}} \) defines the intuitive atomic semantics of transactions.

3 Data-Race Freedom

A data race happens between a pair of conflicting actions, as defined below.
Definition 3.1. A non-transactional request action $\alpha$ and a transactional request action $\alpha'$ conflict if $\alpha$ and $\alpha'$ are executed by different threads and they are read or write actions on the same register, with at least one being a write.

For such actions to form a data race, they should be concurrent. As is standard, we formalize this using a happens-before relation on actions in a history: $hb(H) \subseteq act(H) \times act(H)$. To streamline explanations, we first define DRF in terms of happens-before, and only after this define the latter. For a history $H$ and an index $i$, let $H(i)$ denote the $i$-th action in the sequence $H$.

Definition 3.2. Actions $H(i)$ and $H(j)$ in a history $H$ form a data race, if they conflict and are not related by $hb(H)$ either way. A history $H$ is data-race free (DRF), written $DRF(H)$, if it has no data races.

Definition 3.3. A program $P$ is data-race free (DRF) when executed from a state $s$ with a TM $\mathcal{H}$, written $DRF(P, s, \mathcal{H})$, if $\forall \tau \in \llbracket P \rrbracket\langle \mathcal{H}, s \rangle$. $DRF(\text{history} (\tau))$.

Our goal is to enable programmers to ensure strong atomicity of a program by checking its data-race freedom. However, the notion of DRF depends on the TM $\mathcal{H}$, and we do not want the programmer to have to reason about the actual TM implementation. In Section 5, we give conditions on TM $\mathcal{H}$ under which strong atomicity of a program is guaranteed if it is DRF assuming strong atomicity, i.e., $DRF(P, s, \mathcal{H}_{\text{atomic}})$ for $\mathcal{H}_{\text{atomic}}$ from §2.4. We next define $hb(H)$ and show examples of programs that are racy and race-free under $\mathcal{H}_{\text{atomic}}$.

For a history $H$, we define several relations over $act(H)$, which we explain in the following:

- execution order: $\alpha <_H \alpha'$ iff for some $i, j$ we have $\alpha = H(i)$, $\alpha' = H(j)$ and $i < j$.
- per-thread order $po(H)$: $\alpha <_{po(H)} \alpha'$ iff $\alpha <_H \alpha'$ and actions $\alpha$ and $\alpha'$ are by the same thread.
- restricted per-thread order $xpo(H)$: $\alpha <_{xpo(H)} \alpha'$ iff $\alpha <_H \alpha'$, actions $\alpha$ and $\alpha'$ are by the same thread $t$, and there is a $(\_, t, \text{txbegin})$ action between $\alpha$ and $\alpha'$.
- client order $cl(H)$: $\alpha <_{cl(H)} \alpha'$ iff $\alpha <_H \alpha'$ and $\alpha$ and $\alpha'$ are non-transactional in $H$.
- after-fence order $af(H)$: $\alpha <_{af(H)} \alpha'$ iff $\alpha <_H \alpha'$, $\alpha = (\_, \_, \text{txbegin})$ and $\alpha' = (\_, \_, \text{txbegin})$, i.e., the transaction begins after the fence does (Figure 5(a)).
- before-fence order $bf(H)$: $\alpha <_{bf(H)} \alpha'$ iff $\alpha <_H \alpha'$, $\alpha \in \{\_, \_, \text{committed}, \_, \_, \text{aborted}\}$ and $\alpha' = (\_, \_, \text{fend})$, i.e., the transaction ends before the fence does (Figure 5(b)).
- read-dependency relation $wr_x(H)$ for $x \in \text{Reg}$: $\alpha <_{wr_x(H)} \alpha'$ iff $\alpha = (\_, \_, \text{write}(x, v))$, $\alpha' = (\_, \_, \text{ret}(v))$, and the matching request action for $\alpha'$ is $(\_, \_, \text{read}(x))$.

1The notation $wr$, standing for "write-to-read", is chosen to mirror other kinds of dependencies introduced in §6.

2Note, however, that the commands preceding $\alpha$ in its transaction may not have taken effect by this time: the TM may flush the writes of the

\[ \begin{align*}
\alpha <_{xpo(H)} \alpha' & \iff \alpha <_H \alpha' \\
\alpha <_{cl(H)} \alpha' & \iff \alpha <_H \alpha' \\
\alpha <_{af(H)} \alpha' & \iff \alpha <_H \alpha' \\
\alpha <_{bf(H)} \alpha' & \iff \alpha <_H \alpha' \\
\alpha <_{wr_x(H)} \alpha' & \iff \alpha = (\_, \_, \text{write}(x, v))
\end{align*} \]

Figure 5. Illustration of the fence relations.

\[
\begin{array}{|c|c|}
\hline
\text{line} & \text{code} \\
\hline
11 & \text{:= atomic} \{ \begin{array}{l}
T \text{ do} \\
\quad x := 42; \\
\quad \text{while} \ (\neg t) \\
\quad \text{int} 13 := x; // v''
\end{array} \}
\end{array}
\]

Figure 6. Privatization by agreement outside transactions.

- transactional read-dependency relation $txwr_x(H)$:
  \[ \alpha <_{txwr_x(H)} \alpha' \iff \alpha <_{wr_x(H)} \alpha' , \text{ and } \alpha \text{ and } \alpha' \text{ are transactional}. \]

Definition 3.4. For a history $H$ we let the happens-before relation of $H$ be

\[ hb(H) = \{ po(H) \cup cl(H) \cup af(H) \cup bf(H) \cup \}

\[ \cup_{x \in \text{Reg}}(xpo(H) \cup txwr_x(H)) \}^+. \]

Components of the happens-before describe various forms of synchronization available in our programming language, which we now explain one by one. First, actions by the same thread cannot be concurrent, and thus, we let $po(H) \subseteq hb(H)$. To concentrate on issues related to TM, in this paper we do not consider the integration of transactions into a language with a weak memory model and assume that the underlying non-transactional memory is sequentially consistent. Hence, we do not consider pairs of concurrent non-transactional accesses as races and let $cl(H) \subseteq hb(H)$. This can be used to privatize an object by agreeing on its status outside transactions, as illustrated in Figure 6. There the left-hand-side thread writes to $x$ inside a transaction and then sets the flag $x_{\text{is ready}}$ outside. The right-hand-side thread keeps reading the flag non-transactionally until it is set, and then reads $x$ non-transactionally. This program is DRF under $\mathcal{H}_{\text{atomic}}$ because, in any of its traces, the conflicting write in $T$ and the non-transactional read $v''$ are ordered in happens-before due to the client order between the write in $v$ and the read in $v'$ that causes the do loop to terminate.

We also have $xpo(H) \cup txwr_x(H) \subseteq hb(H)$. Intuitively, this is because, if we have $(\alpha, \alpha') \in txwr_x(H)$, then the commands by the thread of $\alpha$ preceding the transaction of $\alpha$ are guaranteed to have taken effect by the time $\alpha'$ executes.
We state the requirements on a TM we show that the programs in Figure 1 are DRF under $\mathcal{H}_{\text{atomic}}$. Traces of the program may have only a single pair of conflicting actions—the accesses to $x$ in $v$ and $T_2$. For both conflicting actions to occur, $T_2$ has to read $\text{false from } x_{\text{is_private}}$. Since under $\mathcal{H}_{\text{atomic}}$ transactions do not interleave with other transactions or non-transactional accesses, for this $T_1$ has to execute before $T_2$, yielding a history of the form $vT_1T_2$. In this history, we have a read-dependency between the write to $x_{\text{is_private}}$ in $T_1$ and the read from $x_{\text{is_private}}$ in $T_2$. But then the write to $x$ in $v$ happens before the read from $x$ in $T_2$, so that these actions cannot form a race.

Relations $af(H)$ and $bf(H)$ are used to formalize synchronization ensured by transactional fences. Recall that a fence blocks until all active transactions complete, by either committing or aborting. Hence, every transaction either begins after the fence does (and thus the fence does not need to wait for it; Figure S(a)) or ends (including any required clean-up) before the fence does (Figure S(b)). The relations $af(H)$ and $bf(H)$ capture the two respective cases. Note that, as required by Definition 2.1, every transaction has to be related to a fence at least by one of the two relations: a transaction may not span a fence.

Including after-fence and before-fence relations into happens-before ensures that privatization can be done safely given an appropriate placement of fences. To illustrate this, we show that the programs in Figure 1 are DRF under $\mathcal{H}_{\text{atomic}}$ when we place a transactional fence between $x_{\text{is_private}}$. Then $T_2$ has to execute before $T_1$, yielding a history $H$ of the form $T_2T_1a_1a_2v$, where $a_1$ and $a_2$ denote the request and the response actions of the fence. Since $T_2$ occurs before $a_2$ in the history, they are related by the before-fence relation. But then the accesses to $x$ in $T_2$ happen-before the write in $v$ and, therefore, the conflicting actions do not form a race. Finally, the program in Figure 3 is racey, since its traces contain pairs of conflicting actions unordered in happens-before. Inserting fences into this program will not make it DRF.

Our notion of DRF under $\mathcal{H}_{\text{atomic}}$ establishes the condition that a program has to satisfy to be guaranteed strong atomicity. In the next section, we formulate the obligations of its TM counterpart in the DRF contract.

### 4 Strong Opacity

We state the requirements on a TM $\mathcal{H}$ by generalizing the notion of opacity [19, 20], yielding what we call strong opacity. As part of our definition, we require that a history $H$ of a TM $\mathcal{H}$ can be matched by a history $S$ of the atomic TM $\mathcal{H}_{\text{atomic}}$ that “looks similar” to $H$ from the perspective of the program. The similarity is formalized by the following relation $H \subseteq S$, which requires $S$ to be a permutation of $H$ preserving the happens-before relation.

**Definition 4.1.** A history $H_1$ is in the strong opacity relation with a history $H_2$, written $H_1 \subseteq H_2$, if there is a bijection $\theta : \{1, \ldots, |H_1|\} \rightarrow \{1, \ldots, |H_2|\}$ such that:

- $\forall i. H_1(i) = H_2(\theta(i))$, and
- $\forall i,j. i < j \land H_1(i) <_{\text{hb}} H_2(j) \implies \theta(i) < \theta(j)$.

The original definition of opacity requires any history of a TM $\mathcal{H}$ to have a matching history of the atomic TM $\mathcal{H}_{\text{atomic}}$. However, such a requirement would be too strong for our setting: since the TM has no control over non-transactional actions of its clients, histories in $\mathcal{H}$ may be racey, and we do not want to require the TM to guarantee strong atomicity in such cases. Hence, our definition of strong opacity requires only DRF histories to have justifications in $\mathcal{H}_{\text{atomic}}$.

Let $\mathcal{H}_{\text{DRF}} = \{H \in \mathcal{H} | \text{DRF}(H)\}$.

**Definition 4.2.** A TM $\mathcal{H}$ is strongly opaque, written $\mathcal{H}_{\text{DRF}} \subseteq \mathcal{H}_{\text{atomic}}$, if

$$\forall H. H \in \mathcal{H}_{\text{DRF}} \implies \exists S \in \mathcal{H}_{\text{atomic}} \land H \subseteq S.$$

Apart from the restriction to DRF histories, strong opacity and the usual opacity differ in several other ways. First, unlike in the usual opacity, our histories include non-transactional actions, because these can affect the behavior of the TM (e.g., via the idiom of “privatize, modify non-transactionally, publish”, §2.2). Second, instead of preserving happens-before $\text{hb}(H_1)$ in Definition 4.1, the usual opacity requires preserving the program order $\text{po}(H_1)$ and the following real-time order $\text{rt}(H_1)$ on actions: $\alpha <_{\text{rt}(H_1)} \alpha'$ iff $\alpha \in \{\_., \_., \text{committed}, \_., \_., \text{aborted}\}$, $\alpha' = \{\_., \_., \text{txbegin}\}$ and $\alpha <_{\text{rt}} \alpha'$. This orders non-overlapping transactions, with the duration of a transaction determined by the interval from its txbegin action to the corresponding committed or aborted action (or to the end of the history if there is none).

As shown in [16], preserving real-time order is unnecessary if program threads do not have means of communication not reflected in histories. Since we record the actions using both transactional and non-transactional accesses, preserving real-time order is unnecessary for our results. However, we use this order to prove strong opacity by adjusting the proofs of the usual one (§6). Finally, preserving happens-before in Definition 4.1 is required so that we could check DRF assuming strong atomicity, as we explain next.

### 5 The Fundamental Property

We now formalize the Fundamental Property of our DRF notion using observational refinement [7]: if a program is DRF under the atomic TM $\mathcal{H}_{\text{atomic}}$, then any trace of the program under a strongly opaque TM $\mathcal{H}$ has an observationally equivalent trace under the atomic TM $\mathcal{H}_{\text{atomic}}$.
We now develop a method for proving $H | \tau$, where $\tau$ is a TM.

**Definition 5.1.** Traces $\tau$ and $\tau'$ are observationally equivalent, denoted by $\tau \sim \tau'$, if:

$$\left( \forall t \in \text{ThreadID}. \; t | \tau = t | \tau' \right) \land \left( \forall n \in \text{nonTx}. \; \tau | n = \tau' | n \right),$$

where $\tau | n$ denotes the subsequence of $\tau$ containing all actions from non-transactional accesses.

Equivalent traces are considered indistinguishable to the user. In particular, the sequences of non-transactional accesses in equivalent traces (which usually include all input-output) satisfy the same linear-time temporal properties. We lift the equivalence to sets of traces as follows.

**Definition 5.2.** A set of traces $\mathcal{T}$ observationally refines a set of traces $\mathcal{T}'$, written $\mathcal{T} \triangleright= \mathcal{T}'$, if $\forall \tau \in \mathcal{T} \exists \tau' \in \mathcal{T}' \; \tau \sim \tau'$.

**Theorem 5.3 (Fundamental Property).** If $\mathcal{H}$ is a TM such that $\mathcal{H} | \mathcal{DRF} \subseteq \mathcal{H}_{\text{atomic}}$, then

$$\forall P, s. \; \mathcal{DRF}(P, s, \mathcal{H}_{\text{atomic}}) \implies \left[ P \left( \mathcal{H}_{\text{atomic}}, s \right) \leq \left[ P \right] \left( \mathcal{H}, s \right) \right].$$

Theorem 5.3 establishes a contract between the programmer and the TM implementors. The TM implementor has to ensure strong opacity of the TM assuming the program is DRF: $\mathcal{H} | \mathcal{DRF} \subseteq \mathcal{H}_{\text{atomic}}$. The programmer has to ensure the DRF of the program assuming strong atomicity: $\mathcal{DRF}(P, s, \mathcal{H}_{\text{atomic}})$. This contract lets the programmer to check properties of a program assuming strong atomicity ($\left[ P \right]$) and get the guarantee that the properties hold when the program uses the actual TM implementation ($\left[ P \right]$). We have already shown that the expected privatization and publication idioms are DRF under strong atomicity ($\mathcal{H}_{\text{atomic}}$), so that the programmer can satisfy its part of the contract. In the following sections we develop a method for discharging the obligations of the TM.

The proof of Theorem 5.3 follows directly from the following lemma, proved in [25, §B].

**Lemma 5.4.** If $\mathcal{H}$ is a TM such that $\mathcal{H} | \mathcal{DRF} \subseteq \mathcal{H}_{\text{atomic}}$, then:

1. $\forall P, s. \; \mathcal{DRF}(P, s, \mathcal{H}) \implies \left[ P \left( \mathcal{H}, s \right) \leq \left[ P \right] \left( \mathcal{H}_{\text{atomic}}, s \right) \right].$
2. $\forall P, s. \; \mathcal{DRF}(P, s, \mathcal{H}_{\text{atomic}}) \implies \mathcal{DRF}(P, s, \mathcal{H}).$

Part 1 shows that if a program is DRF under the concrete TM $\mathcal{H}$, then it has the expected strongly atomic semantics. It is an adaptation of a result from [7]. Part 2 enables checking DRF using an atomic TM $\mathcal{H}_{\text{atomic}}$ and is a contribution of the present paper. Its proof relies on the fact that strong opacity preserves happens-before (Definition 4.1).

**6 Proving Strong Opacity**

We now develop a method for proving $\mathcal{H} | \mathcal{DRF} \subseteq \mathcal{H}_{\text{atomic}}$. The method builds on a graph characterization of opacity of Guerraoui and Kapalka [20], which was proposed for proving the usual opacity of TMs that do not allow mixed transactional/non-transactional accesses to the same data. The characterization allows checking opacity of a history by checking two properties: consistency of the history and the acyclicity of a certain opacity graph, which we define further in this section.

Consistency captures some very basic properties of read-dependency relation wr$_x$ (for each register $x$) that have to be satisfied by every opaque TM history. Intuitively, in a consistent history every transaction $T$ reading the value of a register $x$ either reads the latest value $T$ itself wrote to $x$ before, or some value written non-transactionally or by a committed or commit-pending transaction.

**Definition 6.1.** A pair of matching request and response actions $(\alpha, \alpha')$ is said to be local to $T \in \text{txns}(H)$, if:

- $\alpha = (\_, \_, \text{read}(x))$ and $(\alpha, \alpha')$ is a contribution of the contract. In the following sections we develop a method for discharging the obligations of the TM.

Thus, local reads from $x$ are preceded by a write to $x$ in the same transaction; local writes to $x$ are followed by a write to $x$ in the same transaction.

**Definition 6.2.** In a history $H$, a read request $\alpha = (\_, \_, \text{read}(x))$ and its matching response $\alpha' = (\_, \_, \text{ret}(v))$ are said to be consistent, if:

- when $(\alpha, \alpha') \in \text{local}(H)$ and performed by a transaction $T$, $v$ is the value written by the most recent write $(\_, \_, \text{write}(x, v))$ preceding the read in $T$;
- when $(\alpha, \alpha') \notin \text{local}(H)$, either there exists a non-local $\beta$ not located in an aborted or live transaction such that $\beta <_{\text{wr}_x(H)} \alpha'$, or there is no such $\beta$ and $v = v_{\text{init}}$.

We also say that a history $H$ is consistent, written $\text{cons}(H)$, if all of its matching read requests and responses are.

We now present the definition of an opacity graph of a history with mixed transactional/non-transactional accesses.

**Definition 6.3.** The opacity graph of a history $H$ is a tuple $G = (N, \text{vis}, \text{HB}, \text{WR}, \text{WW}, \text{RW}, \text{Reg})$, where:

- $N = \text{txns}(H) \cup \text{nontx}(H)$ is the set of nodes.
- $\text{vis} \subseteq N$ is a visibility predicate, such that it holds of all non-transactional accesses and committed transactions and does not hold of all aborted and live transactions.
- $\text{HB} \in \mathcal{P}(N \times N)$ is such that

$$n \xrightarrow{\text{HB}} n' \iff \exists \alpha \in n, \alpha' \in n'. \; \alpha <_{\text{hb}(H)} \alpha'.$$

- $\text{WR} \in \mathcal{P}^\ast \mathcal{P}(N \times N)$ specifies read-dependency relations on nodes: for each $x \in \text{Reg}$,

$$n \xrightarrow{\text{WR}_x} n' \iff n \neq n' \land \exists \alpha \in n, \alpha' \in n'. \; \alpha <_{\text{wr}_x(H)} \alpha'.$$

where the relation on actions wr$_x(H)$ is defined in §3. We require that each node that is read from be visible:

$$\forall n, x. \; n \xrightarrow{\text{WR}_x} \implies \text{vis}(n).$$
We say that a graph $G$ when

Theorem 6.6, we need to make only a minimal adjustment to

history $H$ lifting of the real-time order from §4 to transactions: for a

of the graph, and instead of happens-before, considers the

latter characterization includes only transactions as nodes

a proof of its usual opacity using graph characterization. The

now show that, to prove the strong opacity of a TM using

has to take into account the happens-before relation. We

transactional accesses, ours is more complex: the graph in-

opacity [ 20] for TMs without mixed transactional/non-

Finally, an anti-

specifies when the node

of

dependency

not form a cycle.

The

relation is the lifting of happens-before

where one of the endpoints is a transaction and one is a non-

versions less than or equal to

Theorem 6.6. Let a history $H \in H_{\text{DRF}}$ and an opacity

graph $G = (N, \text{vis}, \text{HB}, \text{WR}, \text{WW}, \text{RW})$ is Graph($H$) be such

that the relation $(\text{HB}; (\text{WR} \cup \text{WW} \cup \text{RW}))$ is irreflexive. If $G$

contains a cycle, then it also contains a cycle over transactions

only with edges from RT $\cup$ WR $\cup$ WW $\cup$ RW.

Thus, the theorem allows us to modularize the proof of the

acyclicity of an opacity graph into: (i) checking the absence of

“small” cycles with a single dependency edge; and (ii)

checking the absence of cycles in the projection of the graph to

transactions, with real-time order replacing happens-before.

The latter acyclicity check is exactly the one required in the

graph characterization of the usual opacity [20]. In the next

section, we show how the theorem enables a simple proof of

strong opacity of a realistic TM, TL2 [12].

The proof of the Theorem 6.6 is given in [25, §B]. Its main

idea lies in the observation that any edge in WR $\cup$ WW $\cup$ RW,

where one of the endpoints is a transaction and one is a non-

transactional access, yields a pair of conflicting actions in

$H$. Since $H$ is DRF, this means that the nodes are related

by HB one way or another, and the irreflexivity of (HB :

(WR $\cup$ WW $\cup$ RW)) means that the dependency edge has to

be covered by HB. Using this, we can transform any cycle in

the graph into one in RT $\cup$ WR $\cup$ WW $\cup$ RW by replacing

segments of edges involving non-transactional accesses by

the real-time order.

7 Case Study: TL2

The TL2 algorithm. The metadata maintained by the TL2

software TM are summarized in Figure 7. For each register $x$, TL2

maintains its value reg[x], version number ver[x] and a

write-lock lock[x]. New version numbers are generated with

the help of a global counter clock, which transactions

advance on commit. For every thread $t$, the TM maintains a

flag active[t], which indicates that the thread $t$ is currently

performing a transaction and is used to implement fences.

TL2 also maintains metadata for each transaction $T$: a read-

set rset[T] of registers $T$ has read from, a write-set wset[T]

of registers and values $T$ intends to write to.

For brevity, we only provide pseudocode for transaction

commits and fences, and describe the initialization, read, and

write informally. When a transaction $T$ starts in a thread

$t$, it sets the flag active[$t$] to true, and stores the value of

clock into a local variable rver[T], which determines $T$’s

read timestamp: TL2 allows $T$ to read registers only with

versions less than or equal to rver[T]. The write of a value

$v$ into a register $x$ simply adds the pair $(x, v)$ to the write-set

wset[T].

Each time $T$ performs a read from a register $x$, it first

checks if it has already performed a write to $x$, in which case

it returns that the value for $x$ from the write-set wset[T].

In other cases, $T$ reads the current value reg[x] and checks
that its version is less than or equal to rver[T]; if not, TL2 aborts the transaction.

Upon a commit, the current transaction \( T \) executes the function \texttt{txcommit} in Figure 7. The commit starts by acquiring locks on each register in the write-set \( \text{wset}[T] \) (lines 11–18). Next, \( T \) fetches-and-increments the value of clock, which it stores into \( \text{wver}[T] \) and uses as the version for the new values \( T \) will write to registers—its \texttt{write timestamp} (line 19). Afterwards, \( T \) ensures that each register \( x \) in \( \text{rset}[T] \) has not been modified during the execution of \( T \) by checking that \( x \)'s version \( \text{ver}[x] \) remains less than or equal to \( \text{rver}[T] \) and that \( x \) is not currently locked (lines 20–26). The transaction then proceeds to write to the registers and release the locks one register at a time (lines 27–30). Finally, \( T \) commits. Upon aborting or committing at lines 18, 26 or 31, \( T \) executes a handler that clears the active[i] flag (not shown in the code).

We consider a simple implementation of transactional fences in lines 33–39 (taken from [17]). The implementation works in two steps: it first determines which transactions the fence should wait for by checking and storing their active flags, and then blocks until the threads performing those transactions clear their active flags.

**Proof overview.** Due to space constraints, we only give an overview of the proof of the strong opacity of TL2. To generate the set \( H_{TL2} \) of all histories of TL2, we consider the most general client of TL2: a program where every thread non-deterministically chooses the commands to execute. The well-formedness conditions on fences from Definition 2.1 can be established with a simple reasoning about the fence function in lines 33–39 independently from the rest of the proof. We prove strong opacity using Theorem 6.5: for every execution of the most general client of TL2 with a DRF history \( H \), we show that \( H \) is consistent and build an opacity graph. To this end, we only need to define a visibility relation \( \text{vis} \) and write-dependencies \( \text{WW} \), as the other components of the graph can be computed from these and \( H \).

The consistency proof and the construction of the graph are inductive in the length of the execution of the most general client. We start with an empty trace, an empty history and an empty graph, and extend them as the executions proceed. Whenever a non-transactional access \( v \) is executed, we add a new visible node to the graph. When \( v \) is a write to a register \( x \), we also append it to the total order \( \text{WW}_x \). Whenever a new transaction \( T \) starts, we add a corresponding invisible node. When \( T \) executes the \texttt{txcommit} function, if it reaches line 27, then we are sure \( T \) is going to commit. At this point we therefore make \( T \) visible and append it to the total order \( \text{WW}_x \) for each register \( x \in \text{wset}[T] \).

We need to show that, whenever the graph is extended with new edges, it stays acyclic. To this end, we use Theorem 6.6 to reduce the acyclicity check to the one required when proving the usual opacity, i.e., checking the absence of cycles over transactions in \( \text{RT} \cup \text{WR} \cup \text{WW} \cup \text{RW} \) (checking the absence of cycles with a single dependency is easier and we omit its description for brevity). In our proof, only graph updates of the read and commit operations of each transaction impose proof obligations.

At every step of the graph construction, we maintain an inductive invariant that helps us prove both consistency of the history and the acyclicity of \( \text{RT} \cup \text{WR} \cup \text{WW} \cup \text{RW} \). Its most important part associates a notion of time with the
edges of the graph based on the read and write timestamps of transactions:

1. \( \forall T', T \xrightarrow{RT} T' \implies \text{rver}[T'] = \bot \lor \exists T' \) where \((\text{vis}(T) \implies \text{wver}[T] = \text{rver}[T']) \land (\neg \text{vis}(T) \implies \text{rver}[T] \leq \text{rver}[T']).\)

2. \( \forall T', T \xrightarrow{WR} T' \implies \text{wver}[T] \leq \text{rver}[T'].\)

3. \( \forall T', T \xrightarrow{RW} T' \implies \text{rver}[T] < \text{wver}[T'].\)

4. \( \forall T, T' \xrightarrow{WW} T' \implies \text{wver}[T] < \text{wver}[T'].\)

Property 1 asserts that, whenever a transaction \( T' \) occurs after a completed transaction \( T \) in the real time, it either has not yet generated a read timestamp \( \text{rver}[T'] \) or it has and \( \text{rver}[T'] \) is greater or equal to \( \text{wver}[T] \) (when \( T \) is visible and, therefore, committed) or \( \text{rver}[T] \) (otherwise). Property 2 asserts that, whenever a transaction \( T' \) reads a value of a register written by a transaction \( T \), the version that \( T' \) assigned to the register may not be greater than the read timestamp of \( T \). This is validated by the check TL2 performs when reading registers. Property 3 asserts that a transaction \( T' \) overwriting the value read by a transaction \( T \) has the write timestamp greater than the read timestamp of \( T \). It holds because, if \( T' \) commits its write after \( T \) reads the previous value of the register, then \( T \) generates its read timestamp before \( T' \) generates its write timestamp. Property 4 follows from the mutual exclusion that TL2 ensures for committing transactions that write the same register \( x \) (using Lock[\( x \)]). Since writes in commit operations occur within a critical section, write dependencies are always consistent with the order on write timestamps.

With the help of the above invariant, we establish that for a path between any transactions \( T \) and \( T' \) in the graph, certain inequalities between their timestamps take place depending on visibility of the two transactions, such as the following:

\[ \text{vis}(T) \land \text{vis}(T') \implies \text{wver}[T] < \text{wver}[T']. \] (1)

Using this and other minor observations, we can demonstrate that graph updates preserve the acyclicity of RT\( \cup \)WR\( \cup \)WW\( \cup \)RW, by showing that a cycle would imply a contradiction involving the timestamps of transactions. As an example of such reasoning, consider a transaction \( T \) executing the txcommit operation. As \( T \) reaches line 27, we mark it as visible and add new write dependencies in the graph. Let us assume that adding \( T' \xrightarrow{WW} T \), where \( T' \) is some transaction, causes a cycle over transactions. Then there must exist a path from \( T \) to \( T' \). Note that \( \text{vis}(T) \) and \( \text{vis}(T') \) both hold, since they are ordered by WW. By (1), \( \text{wver}[T] < \text{wver}[T'] \) holds, because there is a path from \( T \) to \( T' \). On the other hand, Property 4 above gives us \( \text{wver}[T'] < \text{wver}[T], \) since \( T' \xrightarrow{WW} T \) is in the graph. Thus, we have arrived to a contradiction.

8 Related and Future Work

In this paper we have concentrated on one technique for ensuring privatization safety—transactional fences. However, there have been several proposals of alternative techniques (see [21, §4.6.1] for a survey), and in the future, we plan to address these. In particular, some TMs do not require transactional fences for safe privatization [10, 13, 37, 42], even though the programmer still has to follow a certain DRF discipline. Such a discipline has been proposed by Dalessandro and Scott [8, 9], but it did not come with a formal justification, such as our proofs of the Fundamental Property and TM correctness.

Kestor et al. [24] proposed a notion of DRF for TMs that do not support safe privatization and a race-detection tool for this notion. Unlike us, they do not consider transactional fences, so that the only way to safely privatize an object is to agree on its status outside transactions (Figure 6). Our notion of DRF specializes to the one by Kestor et al. if we consider only histories without fences. We hope that, in the future, race-detection tools like the one of Kestor et al. can be adapted to detect our notion of data races.

Lesani et al. [29] proposed a transactional DRF based on TMS [14], a TM consistency criterion. However, as they acknowledge, their proposal does not support privatization.

To the best of our knowledge, a line of work by Abadi et al. was the only one that proposed disciplines for privatization with a formal justification of their safety [3, 4]. However, they did not take into account transactional fences and considered programming disciplines more restrictive than ours. Their static separation [4] ensures strong atomicity by not mixing transactional and non-transactional accesses to the same register. Dynamic separation [3] relaxes this by introducing explicit commands to privatize and publish an object. We believe such disciplines are particular ways of achieving the more general notion of data-race freedom that we adopted.

We have previously proposed a logic for reasoning about programs using RCU [17]. Since transactional fences are similar to RCU, we believe this logic can be adapted to guide programmers in inserting fences to satisfy our notion of DRF.

In this paper we assumed sequential consistency as a baseline non-transactional memory model. However, transactions are being integrated into languages, such as C++, that have weaker memory models [1]. Our definition of a data race is given in the axiomatic style used in the C++ memory model [2]. For this reason, we believe that our results can in the future be adapted to the more complex setting of C++.

Guerraoui et al. [18] considered TMs that provide strong atomicity without making any assumptions about the client program. They formalized the requirement on such TMs as parameterized opacity and proved the impossibility of achieving it on many memory models without instrumenting non-transactional accesses. This result justifies our decision to provide strong atomicity only to DRF programs.

Acknowledgments. We thank Michael Spear and Tingzhe Zhou for discussions about privatization in TM.
References


