# On Parallel Snapshot Isolation and Release/Acquire Consistency

Azalea Raad[1], Ori Lahav[2], and Viktor Vafeiadis[1]

[1] MPI-SWS
[2] Tel Aviv University

**Abstract.** Parallel snapshot isolation (PSI) is a standard transactional consistency model that is used in databases and distributed systems. We argue that PSI is also useful as a formal model for software transactional memory (STM). It has certain advantages over other consistency models, but its formal definition is given declaratively by acyclicity axioms, which most programmers find hard to understand and reason about.
To solve this problem, we develop a simple lock-based reference implementation for PSI built on top of the release-acquire memory model, which is a well-behaved subset of the C/C++11 memory model. We prove that our reference implementation is sound and complete with respect to its higher-level declarative specification.
We further consider an extension of PSI allowing transactional and non-transactional code to interact, and provide a sound and complete reference implementation for the more general setting. Supporting this interaction is necessary for adopting a transactional model in programming languages.

## 1 Introduction

Following the widespread use of transactions in databases, *software transactional memory* (STM) [34,18] has been proposed as a programming language abstraction that can radically simplify the task of writing correct and efficient concurrent programs. It provides the illusion of blocks of code, called *transactions*, executing atomically and in isolation from any other concurrent such blocks.

In theory, STM is great for programmers as it allows them to concentrate on the high-level algorithmic steps of solving a problem and relieves them of such concerns as the low-level details of enforcing mutual exclusion. In practice, however, the situation is far from ideal as the semantics of transactions in the context of non-transactional code is not at all settled. Recent years has seen a plethora of different STM implementations [5,3,19,1,16,2], each providing a slightly different—and often unspecified—semantics to the programmer.

Simple models in the literature are lock-based, such as *global lock atomicity* (GLA) [27] (where a transaction must acquire a global lock prior to execution and release it afterwards) and *disjoint lock atomicity* (DLA) [27] (where a transaction must acquire all locks associated with the locations it accesses prior to execution and release them afterwards), which provide *serialisable* transactions. That is, all transactions appear to have executed atomically one after another in some

total order. The problem with these models is largely their implementation cost, as they impose too much synchronisation between transactions.

The database community has long recognised this performance problem and has developed weaker transactional models that do not guarantee serialisability. The most widely used such model is *snapshot isolation* (SI) [9], implemented by major databases, both centralised (e.g. Oracle and MS SQL Server) and distributed [15,32,29], as well as in STM [1,10,25,24]. In this article, we focus on a closely related model, *parallel snapshot isolation* (PSI) [35], which is known to provide better scalability and availability in large-scale geo-replicated systems. SI and PSI allow conflicting transactions to execute concurrently and to commit successfully, so long as they do not have a write-write conflict. This in effect allows reads of SI/PSI transactions to read from an earlier memory snapshot than the one affected by their writes, and permits outcomes such as the following:

Initially, $x = y = 0$

| | | |
|---|---|---|
| **T1:begin**; | **T2:begin**; | |
| $x := 1$; | $y := 1$; | (SB+txs) |
| $a := y$; *// reads 0* | $b := x$; *// reads 0* | |
| **T1:commit** | **T2:commit** | |

The above is also known as the *write skew anomaly* in the database literature [13]. Such outcomes are analogous to those allowed by weak memory models, such as x86-TSO [33,28] and C11 [8], for non-transactional programs.

In this article, we consider—to the best of our knowledge for the first time—PSI as a possible model for STM, especially in the context of a concurrent language such as C/C++ with a weak memory model. In such contexts, programmers are already familiar with weak behaviours such as that exhibited by SB+txs above.

A key reason why PSI is more suitable for a programming language than SI (or any of the stronger models) is *performance*. This is analogous to why C/C++ adopted non-multi-copy-atomicity (allowing cases where two different threads observe a store by a third thread at different times) as part of their concurrency model. Consider the following "IRIW" (independent reads of independent writes) litmus test:

Initially, $x = y = 0$

| | | | |
|---|---|---|---|
| **T1:begin**; | **T2:begin**; | **T3:begin**; | **T4:begin**; |
| $x := 1$; | $a := x$; *// reads 1* | $c := y$; *// reads 1* | $y := 1$; |
| **T1:commit** | $b := y$; *// reads 0* | $d := x$; *// reads 0* | **T4:commit** |
| | **T2:commit** | **T3:commit** | |

(IRIW+txs)

In the annotated behaviour, transactions T2 and T3 disagree on the relative order of transactions T1 and T4. Under PSI, this behaviour (called the *long fork anomaly*) is allowed, as T1 and T4 are not ordered—they commit in parallel—but it is disallowed under SI. This intuitively means that SI has to impose some ordering guarantees even on transactions that do not access a common location, and can be rather costly in the context of a weakly consistent system.

A second reason why PSI is much more suitable than SI is that it has better properties. One of the key intuitive properties that a programmer might expect of transactions is *monotonicity*. Suppose, in the SB+txs program above, we take the two transactions and split them into four smaller transactions as follows:

Initially, $x = y = 0$

| | | |
|---|---|---|
| **T1:begin**; | **T2:begin**; | |
| $x := 1$; | $y := 1$; | |
| **T1:commit**; | **T2:commit**; | (SB+txs+chop) |
| **T3:begin**; | **T4:begin**; | |
| $a := y$; *// reads 0* | $b := x$; *// reads 0* | |
| **T3:commit** | **T4:commit** | |

One might expect that if the annotated behaviour is allowed in SB+txs, it should also be allowed in SB+txs+chop. This indeed is the case for PSI, but not for SI! In fact, in the extreme case where every transaction contains a single access, SI provides serialisability. Nevertheless, PSI currently has two significant drawbacks, preventing its widespread adoption. We aim to address these in this work.

The first PSI drawback is that its formal semantics can be rather daunting for the uninitiated as it is defined declaratively in terms of acyclicity constraints. What is missing is perhaps a simple lock-based reference implementation of PSI, similar to the lock-based implementations of GLA and DLA, that the programmers can readily understand and reason about. As an added benefit, such an implementation can be viewed as an operational model, forming the basis for developing program logics for reasoning about PSI programs.

Although Cerone et al. [14] proved their declarative PSI specification equivalent to an implementation strategy of PSI in a distributed system with replicated storage over causal consistency, their implementation is not suitable for reasoning about *shared-memory* programs. In particular, it cannot help the programmers determine how transactional and non-transactional accesses may interact.

As our first contribution, in §4 we address this PSI drawback by providing a simple lock-based reference implementation that we prove equivalent to its declarative specification. Typically, one proves that an implementation is *sound* with respect to a declarative specification—i.e. every behaviour observable in the implementation is accounted for in the declarative specification. Here, we also want the other direction, known as *completeness*, namely that every behaviour allowed by the specification is actually possible in the implementation. Having a (simple) complete implementation is very useful for programmers, as it may be easier to understand and experiment with than the declarative specification.

Our reference implementation is built in the *release-acquire* fragment of the C/C++ memory model [20,8,7], using sequence locks [17,22,31,12] to achieve the correct transactional semantics.

The second PSI drawback is that its study so far has not accounted for the subtle effects of non-transactional accesses and how they interact with transactional accesses. While this scenario does not arise in 'closed world' systems such as databases, it is crucially important in languages such as C/C++ and

Java, where one cannot afford the implementation cost of making every access transactional so that it is "strongly isolated" from other concurrent transactions.

Therefore, as our second contribution, in §5 we extend our basic reference implementation to make it robust under uninstrumented non-transactional accesses, and characterise declaratively the semantics we obtain. We call this extended model RPSI (for "robust PSI") and show that it gives reasonable semantics even under scenarios where transactional and non-transactional accesses are mixed.

***Outline*** The remainder of this article is organised as follows. In §2 we present an overview of our contributions and the necessary background information. In §3 we provide the formal model of the C11 release/acquire fragment and describe how we extend it to specify the behaviour of STM programs. In §4 we present our PSI reference implementation (without non-transactional accesses), demonstrating its soundness and completeness against the declarative PSI specification. In §5 we formulate a declarative specification for RPSI as an extension of PSI accounting for non-transactional accesses. We then present our RPSI reference implementation, demonstrating its soundness and completeness against our proposed declarative specification. We conclude and discuss future work in §6.

## 2 Background and Main Ideas

One of the main differences between the specification of database transactions and those of STM is that STM specifications must additionally account for the interactions between *mixed-mode* (both transactional and non-transactional) accesses to the same locations. To characterise such interactions, Blundell et al. [26,11] proposed the notions of *weak* and *strong atomicity*, often referred to as weak and strong isolation. Weak isolation guarantees isolation only amongst transactions: the intermediate state of a transaction cannot affect or be affected by other transactions, but no such isolation is guaranteed with respect to non-transactional code (e.g. the accesses of a transaction may be interleaved by those of non-transactional code.). By contrast, strong isolation additionally guarantees full isolation from non-transactional code. Informally, each non-transactional access is considered as a transaction with a single access. In what follows, we explore the design choices for implementing STMs under each isolation model (§2.1), provide an intuitive account of the PSI model (§2.2), and describe the key requirements for implementing PSI and how we meet them (§2.3).

### 2.1 Implementing Software Transactional Memory

Implementing STMs under either strong or weak isolation models comes with a number of challenges. Implementing strongly isolated STMs requires a conflict detection/avoidance mechanism between transactional and non-transactional code. That is, unless non-transactional accesses are instrumented to adhere to the same access policies, conflicts involving non-transactional code cannot be detected. For instance, in order to guarantee strong isolation under the GLA

model [27] discussed earlier, non-transactional code must be modified to acquire the global lock prior to each shared access and release it afterwards.

Implementing weakly-isolated STMs requires a careful handling of aborting transactions as their intermediate state may be observed by non-transactional code. Ideally, the STM implementation must ensure that the intermediate state of aborting transactions is not leaked to non-transactional code. A transaction may abort either because it failed to commit (e.g. due to a conflict), or because it encountered an explicit abort instruction in the transactional code. In the former case, leaks to non-transactional code can be avoided by pessimistic concurrency control (e.g. locks), pre-empting conflicts. In the latter case, leaks can be prevented either by lazy version management (where transactional updates are stored locally and propagated to memory only upon committing), or by disallowing explicit abort instructions altogether – an approach taken by the (weakly isolated) relaxed transactions of the C++ memory model [5].

As mentioned earlier, our aim in this work is to build an STM with PSI guarantees in the RA fragment of C11. As such, instrumenting non-transactional accesses is not feasible and thus our STM guarantees weak isolation. For simplicity, throughout our development we make a few simplifying assumptions: i) transactions are not nested; ii) the transactional code is without explicit abort instructions (as with the weakly-isolated transactions of C++ [5]); and iii) the locations accessed by a transaction can be statically determined. For the latter, of course, a static over-approximation of the locations accessed suffices for the soundness of our implementations.

### 2.2 Parallel Snapshot Isolation (PSI)

The initial model of PSI introduced in [35] is described informally in terms of a multi-version concurrent algorithm as follows. A transaction $\mathtt{T}$ at a replica $r$ proceeds by taking an initial *snapshot* $S$ of the shared objects in $r$. The execution of $\mathtt{T}$ is then carried out locally: read operations query $S$ and write operations similarly update $S$. Once the execution of $\mathtt{T}$ is completed, it attempts to *commit* its changes to $r$ and may succeed *only if* it is not *write-conflicted*. Transaction $\mathtt{T}$ is write-conflicted if another *committed* transaction $\mathtt{T}'$ has written to a location in $r$ also written to by $\mathtt{T}$, since it recorded its snapshot $S$. If $\mathtt{T}$ fails the conflict check it aborts and may restart the transaction; otherwise, it commits its changes to $r$, at which point its changes become visible to all other transactions that take a snapshot of replica $r$ thereafter. These committed changes are later propagated to other replicas asynchronously.

The main difference between SI and PSI is in the way the committed changes at a replica $r$ are propagated to other sites in the system. Under the SI model, committed transactions are *globally* ordered and the changes at each replica are propagated to others in this global order. This ensures that all concurrent transactions are observed in the same order by all replicas. By contrast, PSI does not enforce a global order on committed transactions: transactional effects are propagated between replicas in *causal* order. This ensures that, if replica $r_1$ commits a message $m$ which is later read at replica $r_2$, and $r_2$ posts a response

$m'$, no replica can see $m'$ without having seen the original message $m$. However, causal propagation allows two replicas to observe concurrent events as if occurring in different orders: if $r_1$ and $r_2$ concurrently commit messages $m$ and $m'$, then replica $r_3$ may initially see $m$ but not $m'$, and $r_4$ may see $m'$ but not $m$. This is best illustrated by the IRIW+txs example in §1.

### 2.3 Towards a Lock-Based Reference Implementation for PSI

While the description of PSI above is suitable for understanding PSI, it is not very useful for integrating the PSI model in a language such as C/++ or Java. From a programmer's perspective, in such languages the various threads directly access the shared memory; they do not access their own replicas, which are loosely related to the replicas of other threads. What we would therefore like is an equivalent description of PSI in terms of unreplicated accesses to shared memory and a synchronisation mechanism such as locks.

In effect, we want a definition similar in spirit to *global lock atomicity* (GLA) [27], which is arguably the simplest TM model, and models committed transactions as acquiring a global mutual exclusion lock, then accessing and updating the data in place, and finally releasing the global lock. Naturally, however, the implementation of PSI cannot be that simple.

A first observation is that PSI cannot be simply implemented over sequentially consistent shared memory.[3] To see this, consider the IRIW+txs program from the introduction. Although PSI allows the annotated behaviour, SC forbids it for the corresponding program without transactions. The point is that under SC, either the $x := 1$ or the $y := 1$ write first reaches memory. Suppose, without loss of generality, that $x := 1$ is written to memory before $y := 1$. Then, the possible atomic snapshots of memory are $x = y = 0$, $x = 1 \wedge y = 0$, and $x = y = 1$. In particular, the snapshot read by T3 is impossible.

To implement PSI we therefore resort to a weaker memory model. Among weak memory models, the "multi-copy-atomic" ones, such as x86-TSO [33,28], SPARC PSO [36,37] and ARMv8-Flat [30], also forbid the weak outcome of IRIW+txs in the same way as SC, and so are unsuitable for our purpose. We thus consider *release-acquire consistency* (RA) [20,8,7], a simple and well-behaved non-multi-copy-atomic model. It is readily available as a subset of the C/C++11 memory model [8] with verified compilation schemes to all major architectures.

RA provides a crucial property that is relied upon in the earlier description of PSI, namely *causality*. In terms of RA, this means that if thread A observes a write $w$ of thread B, then it also observes all the previous writes of thread B as well as any other writes B observed before performing $w$.

A second observation is that using a single lock to enforce mutual exclusion does not work as we need to allow transactions that access disjoint sets of

[3] *Sequential consistency* (SC) [23] is the standard model for shared memory concurrency and defines the behaviours of a multi-threaded program as those arising by executing sequentially some interleaving of the accesses of its constituent threads.

locations to complete in parallel. An obvious solution is to use multiple locks—one per location—as in the *disjoint lock atomicity* (DLA) model [27]. The question remaining is how to implement taking a snapshot at the beginning of a transaction.

A naive attempt is to use reader/writer locks, which allow multiple readers (taking the snapshots) to run in parallel, as long as no writer has acquired the lock. In more detail, the idea is to acquire reader locks for all locations read by a transaction, read the locations and store their values locally, and then release the reader locks. However, as we describe shortly, this approach does not work. Consider the IRIW+txs example. For T2 to get the annotated outcome, it must release its reader lock for $y$ before T4 acquires it. Likewise, since T3 observes $y = 1$, it must acquire its reader lock for $y$ after T4 releases it. By this point, however, it is transitively after the release of the $y$ lock by T2, and so, because of causality, it must have observed all the writes observed by T2 by that point—namely, the $x := 1$ write. In essence, the problem is that reader-writer locks over-synchronise. When two threads acquire the same reader lock, then they synchronise, whereas two read-only transactions should never synchronise in PSI.

To resolve this problem, we use *sequence locks* [17,22,31,12]. Under the sequence locking protocol, each location `x` is associated with a sequence (version) number `vx`, initialised to zero. Each write to `x` increments `vx` before and after its update, provided that `vx` is even upon the first increment. Each read from `x` checks `vx` before and after reading `x`. If both values are the same and even, then there cannot have been any concurrent increments, and the reader must have seen a consistent value. That is, `read(x)` $\triangleq$ `do{v:=vx; s:=x}while(is-odd(v) || vx!=v)`. Under SC, sequence locks are equivalent to reader-writer locks; however, under RA, they are weaker exactly because readers do not synchronise.

**Handling Non-Transactional Accesses** Let us consider what happens if some of the data accessed by a transaction is modified concurrently by an atomic non-transactional write. Since non-transactional accesses do not acquire any locks, the snapshots taken can include values written by non-transactional accesses. The result of the snapshot then depends on the order in which the variables are read. Consider for example the following litmus test:

$$\begin{array}{l||l} x := 1; & \textbf{begin}; \\ y := 1; & a := y; \ \textit{// reads 1} \\ & b := x; \ \textit{// reads 0} \\ & \textbf{commit} \end{array}$$

In our implementation, if the transaction's snapshot reads $y$ before $x$, then the annotated weak behaviour is not possible, because the underlying model (RA) disallows the weak "message passing" behaviour. If, however, $x$ is read before $y$ by the snapshot, then the weak behaviour is possible. In essence, this means that the PSI implementation described so far is of little use, when there are races between transactional and non-transactional code.

Another problem is the lack of *monotonicity*. A programmer might expect that wrapping some code in a transaction block will never yield additional behaviours that were not possible in the program without transactions. Yet, in this example, removing the **begin** and **commit** keywords gets rid of the annotated weak behaviour!

To get monotonicity, it seems that snapshots must read the variables in the same order they are accessed by the transactions. How can this be achieved for transactions that say read $x$, then $y$, and then $x$ again? Or transactions that depending on some complex condition, access first $x$ and then $y$ or vice versa? The key to solving this conundrum is surprisingly simple: *read each variable twice*. In more detail, one takes two snapshots of the locations read by the transaction, and checks that both snapshots return the same values for each location. This ensures that every location is read both before and after every other location in the transaction, and hence all the high-level happens-before orderings in executions of the transactional program are also respected by its implementation.

There is however one caveat: since equality of values (and version numbers) is used to determine whether the two snapshots are the same, we will miss cases where different non-transactional writes to a variable write the same value. In our formal development (see §5), we thus assume that if, in an execution two writes to the same location write the same value, then both are transactional. This assumption cannot be lifted without instrumenting non-transactional accesses.

## 3 The Release-Acquire Memory Model for STM

We present the notational conventions used in the remainder of this article and proceed with the declarative model of the *release-acquire* (RA) fragment [20] of the C11 memory model [8], in which we implement our STM. In §3.1 we describe how we extend this formal model to specify the behaviour of STM programs.

***Notation*** Given a relation $\mathsf{r}$ on a set $A$, we write $\mathsf{r}^?$, $\mathsf{r}^+$ and $\mathsf{r}^*$ for the reflexive, transitive and reflexive-transitive closure of $\mathsf{r}$, respectively. We write $\mathsf{r}^{-1}$ for the inverse of $\mathsf{r}$; $\mathsf{r}|_A$ for $\mathsf{r} \cap A^2$; $[A]$ for the identity relation on $A$, i.e. $\{(a,a) \mid a \in A\}$; $\mathsf{irreflexive}(r)$ for $\neg\exists a.\ (a,a) \in r$; and $\mathsf{acyclic}(\mathsf{r})$ for $\mathsf{irreflexive}(\mathsf{r}^+)$. Given two relations $\mathsf{r}_1$ and $\mathsf{r}_2$, we write $\mathsf{r}_1;\mathsf{r}_2$ for their (left) relational composition, i.e. $\{(a,b) \mid \exists c.\ (a,c) \in \mathsf{r}_1 \wedge (c,b) \in \mathsf{r}_2\}$. Lastly, when $\mathsf{r}$ is a strict partial order, we write $\mathsf{r}|_{\mathrm{imm}}$ for the *immediate* edges in $\mathsf{r}$: $\{(a,b) \in \mathsf{r} \mid \neg\exists c.\ (a,c) \in \mathsf{r} \wedge (c,b) \in \mathsf{r}\}$.

The RA model is given by the fragment of the C11 memory model, where all read accesses are acquire (`acq`) reads, all writes are release (`rel`) writes, and all atomic updates (i.e. RMWs) are acquire-release (`acqrel`) updates. The semantics of a program under RA is defined as a set of *consistent executions*.

**Definition 1 (Executions in RA).** Assume a finite set of *locations* Loc; a finite set of *values* Val; and a finite set of *thread identifiers* Tid. Let $x, y, z$ range over locations, $v$ over values and $\tau$ over thread identifiers. An *RA execution graph of an STM implementation*, $G$, is a tuple of the form $(E, \mathsf{po}, \mathsf{rf}, \mathsf{mo})$ with its nodes given by $E$ and its edges given by the $\mathsf{po}$, $\mathsf{rf}$ and $\mathsf{mo}$ relations such that:

- $E \subset \mathbb{N}$ is a finite set of *events*, and is accompanied with the functions $\mathtt{tid}(.) : E \to \textsc{Tid}$ and $\mathtt{lab}(.) : E \to \textsc{Label}$, returning the thread identifier and the label of an event, respectively. We typically use $a$, $b$, and $e$ to range over events. The label of an event is a tuple of one of the following three forms: i) $(\mathtt{R}, x, v)$ for *read* events; ii) $(\mathtt{W}, x, v)$ for *write* events; or iii) $(\mathtt{U}, x, v, v')$ for *update* events. The $\mathtt{lab}(.)$ function induces the functions $\mathtt{typ}(.)$, $\mathtt{loc}(.)$, $\mathtt{val_r}(.)$ and $\mathtt{val_w}(.)$ that respectively project the type ($\mathtt{R}$, $\mathtt{W}$ or $\mathtt{U}$), location, and read/written values of an event, where applicable. The set of *read events* is denoted by $R \triangleq \{e \in E \mid \mathtt{typ}(e) \in \{\mathtt{R}, \mathtt{U}\}\}$; similarly, the set of *write events* is denoted by $W \triangleq \{e \in E \mid \mathtt{typ}(e) \in \{\mathtt{W}, \mathtt{U}\}\}$ and the set of *update events* is denoted by $U \triangleq R \cap W$.
  We further assume that $E$ always contains a set $E_0$ of initialisation events consisting of a write event with label $(\mathtt{W}, x, 0)$ for every $x \in \textsc{Loc}$.
- $\mathsf{po} \subseteq E \times E$ denotes the *'program-order' relation*, defined as a disjoint union of strict total orders, each orders the events of one thread, together with $E_0 \times (E \setminus E_0)$ that places the initialisation events before any other event.
- $\mathsf{rf} \subseteq W \times R$ denotes the *'reads-from' relation*, defined as a relation between write and read events of the same location and value; it is total and functional on reads, i.e. every read event is related to exactly one write event;
- $\mathsf{mo} \subseteq W \times W$ denotes the *'modification-order' relation*, defined as a disjoint union of strict orders, each of which totally orders the write events to one location.

We often use "$G.$" as a prefix to project the various components of $G$ (e.g. $G.E$). Given a relation $\mathtt{r} \subseteq E \times E$, we write $\mathtt{r}_{loc}$ for $\mathtt{r} \cap \{(a, b) \mid \mathtt{loc}(a) = \mathtt{loc}(b)\}$. Analogously, given a set $A \subseteq E$, we write $A_x$ for $A \cap \{a \mid \mathtt{loc}(a) = x\}$. Lastly, given the rf and mo relations, we define the 'reads-before' relation $\mathsf{rb} \triangleq \mathsf{rf}^{-1}; \mathsf{mo}$.

Executions of a given program represent traces of shared memory accesses generated by the program. We only consider "partitioned" programs of the form $\|_{\tau \in \textsc{Tid}} c_\tau$, where Tid is a finite set of thread identifiers, the $\|$ denotes parallel composition, and each $c_i$ is a sequential program.

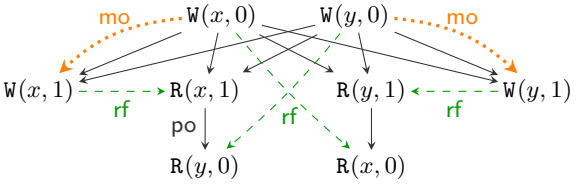


Fig. 1: An RA-consistent execution of a transaction-free variant of IRIW+txs in §1, with program outcome $a = c = 1$ and $b = d = 0$.

The set of executions associated with a given program is then defined by induction over the structure of sequential programs. We do not define this construction formally as it depends on the syntax of the implementation programming language. Each execution of a program $P$ has a particular program *outcome*, prescribing the final values of local variables in each thread (see example in Fig. 1).

In this initial stage, the execution outcomes are unrestricted in that there are no constraints on the rf and mo relations. The restrictions on rf and mo and

thus the permitted outcomes of a program are determined by defining the set of *consistent* executions.

**Definition 2 (RA-consistency).** A program execution $G$ is *RA-consistent*, written $\mathsf{consistent}(G)$, if $\mathsf{acyclic}(\mathsf{hb}_{loc} \cup \mathsf{mo} \cup \mathsf{rb})$ holds, where $\mathsf{hb} \triangleq (\mathsf{po} \cup \mathsf{rf})^+$ denotes the 'RA-happens-before' relation.

Among all executions of a given program $P$, only the *RA-consistent* ones define the allowed outcomes of $P$.

### 3.1 Software Transactional Memory in RA: Specification

Our goal in this section is to develop a declarative framework that allows us to specify the behaviour of mixed-mode STM programs under weak isolation guarantees. Whilst the behaviour of transactional code is dictated by the particular isolation model considered (e.g. PSI), the behaviour of non-transactional code and its interaction with transactions is guided by the underlying memory model. As we build our STM in the RA fragment of C11, we assume the behaviour of non-transactional code to conform to the RA memory model. More concretely, we build our specification of a program $P$ such that i) in the absence of transactional code, the behaviour of $P$ is as defined by the RA memory model; ii) in the absence of non-transactional code, the behaviour of $P$ is as defined by the PSI model. We proceed with the definition of our STM specification executions.

**Definition 3 (Specification executions).** Assume a finite set of *transaction identifiers* $\textsc{TxId}$. An *execution graph of an STM specification*, $\Gamma$, is a tuple of the form $(E, \mathsf{po}, \mathsf{rf}, \mathsf{mo}, T)$ where:

- $E \triangleq R \cup W \cup B \cup E$, denotes the set of *events* with $R$ and $W$ defined as the sets of read, write and update events as described above; and the $B$ and $E$ respectively denote the set of events marking the *beginning* and *end of transactions*. For each event $a \in B \cup E$, the $\mathtt{lab}(.)$ function is extended to return $\mathtt{B}$ when $a \in B$, and $\mathtt{E}$ when $a \in E$. The $\mathtt{typ}(.)$ function is accordingly extended to return a type in $\{\mathtt{R}, \mathtt{W}, \mathtt{U}, \mathtt{B}, \mathtt{E}\}$, whilst the remaining functions are extended to return default (dummy) values for events in $B \cup E$.
- $\mathsf{po}$, $\mathsf{rf}$ and $\mathsf{mo}$ denote the *'program-order'*, *'reads-from'* and *'modification-order' relations* as described above;
- $T \subseteq E$ denotes the set of *transactional events* with $B \cup E \subseteq T$. For transactional events in $T$, event labels are extended to carry an additional component, namely the associated transaction identifier. As such, a specification graph is additionally accompanied with the function $\mathtt{tx}(.) : T \to \textsc{TxId}$, returning the transaction identifier of transactional events. The derived *'same-transaction' relation*, $\mathsf{st} \in T \times T$, is the equivalence relation given by $\mathsf{st} \triangleq \{(a, b) \in T \times T \mid \mathtt{tx}(a) = \mathtt{tx}(b)\}$.

We write $T/\mathsf{st}$ for the set of equivalence classes of $T$ induced by $\mathsf{st}$; $[a]_{\mathsf{st}}$ for the equivalence class that contains $a$; and $T_\xi$ for the equivalence class of transaction $\xi \in \textsc{TxId}$: $T_\xi \triangleq \{a \mid \mathtt{tx}(a){=}\xi\}$. We write $NT$ for non-transactional events: $NT \triangleq E \setminus T$. We often use "$\Gamma.$" as a prefix to project the $\Gamma$ components.

***Specification consistency*** The consistency of specification graphs is model-specific in that it is dictated by the guarantees provided by the underlying model. In the upcoming sections, we present two consistency definitions of PSI in terms of our specification graphs that lack cycles of certain shapes. In doing so, we often write $\mathsf{r}_\mathsf{T}$ for lifting a relation $\mathsf{r} \subseteq E \times E$ to transaction classes: $\mathsf{r}_\mathsf{T} \triangleq \mathsf{st}; (\mathsf{r} \setminus \mathsf{st}); \mathsf{st}$. Analogously, we write $\mathsf{r}_\mathsf{I}$ to restrict $\mathsf{r}$ to the internal events of a transaction: $\mathsf{r} \cap \mathsf{st}$.

***Comparison to dependency graphs*** Adya et al. proposed *dependency graphs* for declarative specification of transactional consistency models [6,4]. Dependency graphs are similar to our specification graphs in that they are constructed from a set of nodes and a set of edges (relations) capturing certain dependencies. However, unlike our specification graphs, the nodes in dependency graphs denote entire transactions and not individual events. In particular, Adya et al. propose three types of dependency edges: i) a *read dependency* edge, $T_1 \overset{WR}{\rightarrow} T_2$, denotes that transaction $T_2$ reads a value written by $T_1$; ii) a *write dependency* edge $T_1 \overset{WW}{\rightarrow} T_2$ denotes that $T_2$ overwrites a value written by $T_1$; and iii) an *anti-dependency* edge $T_1 \overset{RW}{\rightarrow} T_2$ denotes that $T_2$ overwrites a value read by $T_1$. Adya's formalism does not allow for *non-transactional* accesses and it thus suffices to define the dependencies of an execution as edges between transactional classes. In our specification graphs however, we account for both transactional and non-transactional accesses and thus define our relational dependencies between individual events of an execution. However, when we need to relate an entire transaction to another with relation $\mathsf{r}$, we use the transactional lift ($\mathsf{r}_\mathsf{T}$) defined above. In particular, Adya's dependency edges correspond to ours as follows. Informally, the *WR* corresponds to our $\mathsf{rf}_\mathsf{T}$; the *WW* corresponds to our $\mathsf{mo}_\mathsf{T}$; and the *RW* corresponds to our $\mathsf{rb}_\mathsf{T}$. Adya's dependency graphs have been used to develop declarative specifications of the PSI consistency model [13]. In §4, we revisit this model, redefine it as specification graphs in our settings, and develop a reference lock-based implementation that is sound and complete with respect to this abstract specification. The model in [13] does not account for non-transactional accesses. To remedy this, later in §5, we develop a declarative specification of PSI that allows for both transactional and non-transactional accesses. We then develop a reference lock-based implementation that is sound and complete with respect to our proposed model.

## 4 Parallel Snapshot Isolation (PSI)

We present a declarative specification of PSI (4.1), and develop a lock-based reference implementation of PSI in the RA fragment (§4.2). We then demonstrate that our implementation is both sound (§4.3) and complete (§4.4) with respect to the PSI specification. Note that the PSI model in this section accounts for transactional code only; that is, throughout this section we assume that $\Gamma.E = \Gamma.T$. We lift this assumption later in §5.

### 4.1 A Declarative Specification of PSI STMs in RA

In order to formally characterise the weak behaviour and anomalies admitted by PSI, Cerone and Gotsman [14,13] formulated a declarative PSI specification. (In fact, they provide two equivalent specifications: one using dependency graphs proposed by Adya et al. [6,4]; and the other using abstract executions.) As is standard, they characterise the set of executions admitted under PSI as graphs that lack certain cycles. We present an equivalent declarative formulation of PSI, adapted to use our notation as discussed in §3. It is straightforward to verify that our definition coincides with the dependency graph specification in [14].

***PSI consistency*** A PSI execution graph $\mathit{\Gamma}{=}(E, \mathsf{po}, \mathsf{rf}, \mathsf{mo}, T)$ is *consistent*, written $\mathsf{psi\text{-}consistent}(\mathit{\Gamma})$, if the following hold:

- $\mathsf{rf}_\mathsf{I} \cup \mathsf{mo}_\mathsf{I} \cup \mathsf{rb}_\mathsf{I} \subseteq \mathsf{po}$ (INT)
- $\mathsf{irreflexive}((\mathsf{po}_\mathsf{T} \cup \mathsf{rf}_\mathsf{T} \cup \mathsf{mo}_\mathsf{T})^+; \mathsf{rb}_\mathsf{T}^?)$ (EXT)

Informally, INT ensures the consistency of each transaction internally, while EXT provides the synchronisation guarantees among transactions. In particular, we note that the two conditions together ensure that if two read events in the same transaction read from the same location $x$, and no write to $x$ is po-between them, then they must read from the same write (known as 'internal read consistency').

Next, we provide an alternative formulation of PSI-consistency that is closer in form to RA-consistency. This formulation is the basis of our extension in §5 with non-transactional accesses.

**Lemma 1.** *A PSI execution graph $\mathit{\Gamma} = (E, \mathsf{po}, \mathsf{rf}, \mathsf{mo}, T)$ is consistent if and only if* $\mathsf{acyclic}(\mathsf{psi\text{-}hb}_{loc} \cup \mathsf{mo} \cup \mathsf{rb})$ *holds, where* psi-hb *denotes the* 'PSI-happens-before' *relation, defined as* $\mathsf{psi\text{-}hb} \triangleq (\mathsf{po} \cup \mathsf{rf} \cup \mathsf{rf}_\mathsf{T} \cup \mathsf{mo}_\mathsf{T})^+$.

*Proof. The full proof is provided in the technical appendix.*

Note that this acyclicity condition is rather close to that of RA-consistency definition presented in §3, with the sole difference being the definition of 'happens-before' relation by replacing hb with psi-hb. The relation psi-hb is a strict extension of hb with $\mathsf{rf}_\mathsf{T} \cup \mathsf{mo}_\mathsf{T}$, which captures additional synchronisation guarantees resulting from transaction orderings, as described shortly. As in RA-consistency, the po and rf are included in the 'PSI-happens-before' relation psi-hb. Additionally, the $\mathsf{rf}_\mathsf{T}$ and $\mathsf{mo}_\mathsf{T}$ also contribute to psi-hb.

Intuitively, the $\mathsf{rf}_\mathsf{T}$ corresponds to synchronisation due to causality between transactions. A transaction $\mathtt{T}_1$ is causally-ordered before transaction $\mathtt{T}_2$, if $\mathtt{T}_1$ writes to $\mathtt{x}$ and $\mathtt{T}_2$ later (in 'happens-before' order) reads $\mathtt{x}$. The inclusion of $\mathsf{rf}_\mathsf{T}$ ensures that $\mathtt{T}_2$ cannot read from $\mathtt{T}_1$ without observing its entire effect. This in turn ensures that transactions exhibit an atomic 'all-or-nothing' behaviour. In particular, transactions cannot mix-and-match the values they read. For instance, if $\mathtt{T}_1$ writes to both $x$ and $y$, transaction $\mathtt{T}_2$ may not read the value of $x$ from $\mathtt{T}_1$ but read the value of $y$ from an earlier (in 'happens-before' order) transaction $\mathtt{T}_0$.

| | |
|---|---|
| **begin**; | `for(x in WS) { lock vx; }` |
| `T;` | `snapshot(RS); T';` |
| **end** | `for(x in WS) { unlock vx; }` |

where `v[]` and `s[]` are thread-local arrays,
and

```
          vx ≜ x+1
   lock vx ≜ do {
               v[x] := vx;
               if (is-odd(v[x])) { continue; }
               b := CAS(vx, v[x], v[x]+1)
             } while (!b)
snapshot(RS) ≜ do {
               for(x in RS) {
                 a := vx;
                 if (is-odd(a) && !(x in WS)) {continue}
                 if (!(x in WS)) { v[x] := a }
                 s[x] := x;
               } b := true;
               for(x in RS) {
                 if (!b) { break; }
                 a := vx; b := (v[x] == a);
             } } while (!b)
 unlock vx ≜ vx := v[x] + 2
        T' ≜ Tr[x := a; s[x] := a/x := a]   for x := a in Tr
        Tr ≜ T[a := s[x]/a := x]   for a := x in T
```

Fig. 2: Lock-based PSI implementation of `T` given `RS` and `WS`

The $\mathsf{mo}_\mathsf{T}$ corresponds to synchronisation due to conflicts between transactions. Its inclusion enforces the write-conflict-freedom of PSI transactions. In other words, if two transactions $\mathtt{T}_1$ and $\mathtt{T}_2$ both write to the same location $x$ via events $w_1$ and $w_2$ such that $w_1 \overset{\mathsf{mo}}{\rightarrow} w_2$, then $\mathtt{T}_1$ must commit before $\mathtt{T}_2$, and thus the entire effect of $\mathtt{T}_1$ must be visible to $\mathtt{T}_2$.

### 4.2 A Lock-based PSI Implementation in RA

We present an operational model of PSI that is both sound and complete with respect to the declarative semantics in §4.1. To this end, in Fig. 2 we develop a pessimistic (lock-based) reference implementation of PSI using sequence locks [17,22,31,12], referred to as *version locks* in our implementation. In order to avoid taking a snapshot of the *entire* memory and thus decrease the locking overhead, we assume that a transaction `T` is supplied with its *read set*,

RS, containing those locations that are read by T. Similarly, we assume T to be supplied with its *write set*, WS, containing the locations updated by T.[4]

The implementation of T proceeds by exclusively acquiring the version locks on all locations in its write set. It then obtains a snapshot s of the locations in its read set by inspecting their version locks, as described shortly, and subsequently recording their values in s. Once a snapshot is recorded, the execution of the transaction proceeds as follows. Each read operation consults the local snapshot s; each write operation updates the memory eagerly (in-place) and subsequently updates its local snapshot to ensure correct lookup for future reads, as required by internal read consistency. Once the execution of T is concluded, the version locks on the write set are released. Observe that as the writer locks are acquired pessimistically, we do not need to check for write-conflicts in the implementation.

To facilitate our locking implementation, we assume that each location x is associated with a version lock at address x+1, written vx. The value held by a version lock vx may be in one of two categories: i) an even number, denoting that the lock is free; or ii) an odd number, denoting that the lock is exclusively held by a writer. For a transaction to write to a location x in its write set WS, the x version lock (vx) must be acquired exclusively by calling `lock vx`. Each call to `lock vx` reads the value of vx and stores it in v[x]. It then checks if the value read is even (vx is free) and if so it atomically increments it by 1 (with a 'compare-and-swap' operation), thus changing the value of vx to an odd number and acquiring it exclusively; otherwise it repeats this process until the version lock is successfully acquired. Conversely, each call to `unlock vx` updates the value of vx to v[x]+2, restoring the value of vx to an even number and thus releasing it. Note that deadlocks can be avoided by imposing an ordering on locks and ensuring their in-order acquisition by all transactions. For simplicity however, we have elided this step as we are not concerned with progress or performance issues here and our main objective is a reference implementation of PSI in RA.

Analogously, for a transaction to read from the locations in its read set RS, it must record a snapshot of their values by calling `snapshot(RS)`. To obtain a snapshot of location x, the transaction must ensure that x is not currently being written to by another transaction. It thus proceeds by reading the value of vx and recording it in v[x]. If vx is free (the value read is even) or x is in its write set WS, the value of x can be freely read and tentatively stored in s[x]. In the latter case, the transaction has already acquired the exclusive lock on vx and is thus safe in the knowledge that no other transaction is currently updating x. Once a tentative snapshot of all locations is obtained, the transaction must *validate* it by ensuring that it reflects the values of the read set at a single point in time. To do this, it revisits the version locks, inspecting whether their values have changed (by checking them against the recorded versions in v) since it recorded its snapshot. If so, then an intermediate update has intervened, potentially invalidating the obtained snapshot; the transaction thus restarts the snapshot process. Otherwise, the snapshot is successfully validated and returned in s.

[4] A conservative estimate of RS and WS can be obtained by simple syntactic analysis.

### 4.3 Implementation Soundness

The PSI implementation in Fig. 2 is *sound*: for each RA-consistent implementation graph $G$, a corresponding specification graph $\Gamma$ can be constructed such that $\mathsf{psi\text{-}consistent}(\Gamma)$ holds. In what follows we state our soundness theorem and briefly describe our construction of consistent specification graphs. We refer the reader to the technical appendix for the full soundness proof.

**Theorem 1 (Soundness).** *For all RA-consistent implementation graphs $G$ of the implementation in Fig. 2, there exists a PSI-consistent specification graph $\Gamma$ of the corresponding transactional program that has the same program outcome.*

**Constructing Consistent Specification Graphs** Observe that given an execution of our implementation with $t$ transactions, the trace of each transaction $i \in \{1 \cdots t\}$ is of the form $\theta_i = Ls_i \overset{\mathsf{po}}{\to} FS_i \overset{\mathsf{po}}{\to} S_i \overset{\mathsf{po}}{\to} Ts_i \overset{\mathsf{po}}{\to} Us_i$, where $Ls_i$, $FS_i$, $S_i$, $Ts_i$ and $Us_i$ respectively denote the sequence of events acquiring the version locks, attempting but failing to obtain a valid snapshot, recording a valid snapshot, performing the transactional operations, and releasing the version locks. For each transactional trace $\theta_i$ of our implementation, we thus construct a corresponding trace of the specification as $\theta'_i = B_i \overset{\mathsf{po}}{\to} Ts'_i \overset{\mathsf{po}}{\to} E_i$, where $B_i$ and $E_i$ denote the transaction begin and end events ($\mathtt{lab}(B_i){=}\mathtt{B}$ and $\mathtt{lab}(E_i){=}\mathtt{E}$). When $Ts_i$ is of the form $t_1 \overset{\mathsf{po}}{\to} \cdots \overset{\mathsf{po}}{\to} t_n$, we construct $Ts'_i$ as $t'_1 \overset{\mathsf{po}}{\to} \cdots \overset{\mathsf{po}}{\to} t'_n$ with each $t'_j$ defined either as $t'_j \triangleq (\mathtt{R}, \mathtt{x}, v)$ when $t_j = (\mathtt{R}, \mathtt{s[x]}, v)$ (i.e. the corresponding implementation event is a read event); or as $t'_j \triangleq (\mathtt{W}, \mathtt{x}, v)$ when $t_j{=}(\mathtt{W}, \mathtt{x}, v) \overset{\mathsf{po}}{\to} (\mathtt{W}, \mathtt{s[x]}, v)$.

For each specification trace $\theta'_i$ we construct the 'reads-from' relation as:

$$\mathsf{RF}_i \triangleq \left\{ (w, t'_j) \;\middle|\; \begin{array}{l} t'_j \in Ts'_i \wedge \exists \mathtt{x}, v.\ t'_j{=}(\mathtt{R}, \mathtt{x}, v) \wedge w{=}(\mathtt{W}, \mathtt{x}, v) \\ \wedge (w \in Ts'_i \Rightarrow w \overset{\mathsf{po}}{\to} t'_j \wedge \\ \qquad\qquad (\forall e \in Ts'_i.\ w \overset{\mathsf{po}}{\to} e \overset{\mathsf{po}}{\to} t'_j \Rightarrow (\mathtt{loc}(e){\neq}\mathtt{x} \vee e{\notin} W))) \\ \wedge (w \notin Ts'_i \Rightarrow (\forall e{\in} Ts'_i.\ (e \overset{\mathsf{po}}{\to} t'_j \Rightarrow (\mathtt{loc}(e) \neq \mathtt{x} \vee e \notin W)) \\ \qquad\qquad \wedge \exists r' \in S_i.\ \mathtt{loc}(r'){=}\mathtt{x} \wedge (w, r') \in G.\mathsf{rf}) \end{array} \right\}$$

That is, we construct our graph such that each read event $t'_j$ from location $\mathtt{x}$ in $Ts'_i$ either i) is preceded by a write event $w$ to $\mathtt{x}$ in $Ts'_i$ without an intermediate write in between them and thus 'reads-from' $w$ (lines two and three); or ii) is not preceded by a write event in $Ts'_i$ and thus 'reads-from' the write event $w$ from which the initial snapshot read $r'$ in $S_i$ obtained the value of $\mathtt{x}$ (last two lines).

Given a consistent implementation graph $G = (E, \mathsf{po}, \mathsf{rf}, \mathsf{mo})$, we construct a consistent specification graph $\Gamma = (E, \mathsf{po}, \mathsf{rf}, \mathsf{mo}, T)$ such that:

- $\Gamma.E \triangleq \bigcup_{i \in \{1 \cdots t\}} \theta'_i.E$ – the events of $\Gamma.E$ is the union of events in each transaction trace $\theta'_i$ of the specification constructed as above;
- $\Gamma.\mathsf{po} \triangleq G.\mathsf{po}|_{\Gamma.E}$ – the $\Gamma.\mathsf{po}$ is that of $G.\mathsf{po}$ limited to the events in $\Gamma.E$;
- $\Gamma.\mathsf{rf} \triangleq \bigcup_{i \in \{1 \cdots t\}} \mathsf{RF}_i$ – the $\Gamma.\mathsf{rf}$ is the union of $\mathsf{RF}_i$ relations defined above;
- $\Gamma.\mathsf{mo} \triangleq G.\mathsf{mo}|_{\Gamma.E}$ – the $\Gamma.\mathsf{mo}$ is that of $G.\mathsf{mo}$ limited to the events in $\Gamma.E$;
- $\Gamma.T \triangleq \Gamma.E$, where for each $e \in \Gamma.T$, we define $\mathtt{tx}(e) = i$ when $e \in \theta'_i$.

### 4.4 Implementation Completeness

The PSI implementation in Fig. 2 is *complete*: for each consistent specification graph $\Gamma$ a corresponding implementation graph $G$ can be constructed such that $\mathsf{consistent}(G)$ holds. We next state our completeness theorem and describe our construction of consistent implementation graphs. We refer the reader to the technical appendix for the full completeness proof.

**Theorem 2 (Completeness).** *For all PSI-consistent specification graphs $\Gamma$ of a transactional program, there exists an RA-consistent execution graph $G$ of the implementation in Fig. 2 that has the same program outcome.*

**Constructing Consistent Implementation Graphs** In order to construct an execution graph of the implementation $G$ from the specification $\Gamma$, we follow similar steps as those in the soundness construction, in reverse order. More concretely, given each trace $\theta'_i$ of the specification, we construct an analogous trace of the implementation by inserting the appropriate events for acquiring and inspecting the version locks, as well as obtaining a snapshot. For each transaction class $T_i \in T/\mathsf{st}$, we must first determine its read and write sets and subsequently decide the order in which the version locks are acquired (for locations in the write set) and inspected (for locations in the read set). This then enables us to construct the 'reads-from' and 'modification-order' relations for the events associated with version locks.

Given a consistent execution graph of the specification $\Gamma = (E, \mathsf{po}, \mathsf{rf}, \mathsf{mo}, T)$, and a transaction class $T_i \in \Gamma.T/\mathsf{st}$, we write $\mathtt{WS}_{T_i}$ for the set of locations written to by $T_i$. That is, $\mathtt{WS}_{T_i} \triangleq \bigcup_{e \in T_i \cap W} \mathtt{loc}(e)$. Similarly, we write $\mathtt{RS}_{T_i}$ for the set of locations read from by $T_i$, *prior to* being written to by $T_i$. For each location $\mathtt{x}$ read from by $T_i$, we additionally record the first read event in $T_i$ that retrieved the value of $\mathtt{x}$. That is,

$$\mathtt{RS}_{T_i} \triangleq \left\{ (\mathtt{x}, r) \,\middle|\, r \in T_i \cap R_{\mathtt{x}} \wedge \neg \exists e \in T_i \cap E_{\mathtt{x}}.\ e \overset{\mathsf{po}}{\to} r \right\}$$

Note that transaction $T_i$ may contain several read events reading from $\mathtt{x}$, prior to subsequently updating it. However, the internal-read-consistency property ensures that all such read events read from the same write event. As such, as part of the read set of $T_i$ we record the first such read event (in program-order).

Determining the ordering of lock events hinges on the following observation. Given a consistent execution graph of the specification $\Gamma = (E, \mathsf{po}, \mathsf{rf}, \mathsf{mo}, T)$, let for each location $\mathtt{x}$ the total order $\mathsf{mo}$ be given as: $w_1 \overset{\mathsf{mo}|_{\mathsf{imm}}}{\to} \cdots \overset{\mathsf{mo}|_{\mathsf{imm}}}{\to} w_{n_{\mathtt{x}}}$. Observe that this order can be broken into adjacent segments where the events of each segment belong to the *same* transaction. That is, given the transaction classes $\Gamma.T/\mathsf{st}$, the order above is of the following form where $T_1, \cdots, T_m \in \Gamma.T/\mathsf{st}$ and for each such $T_i$ we have $\mathtt{x} \in \mathtt{WS}_{T_i}$ and $w_{(i,1)} \cdots w_{(i,n_i)} \in T_i$:

$$\underbrace{w_{(1,1)} \overset{\mathsf{mo}|_{\mathsf{imm}}}{\to} \cdots \overset{\mathsf{mo}|_{\mathsf{imm}}}{\to} w_{(1,n_1)}}_{T_1} \overset{\mathsf{mo}|_{\mathsf{imm}}}{\to} \cdots \overset{\mathsf{mo}|_{\mathsf{imm}}}{\to} \underbrace{w_{(m,1)} \overset{\mathsf{mo}|_{\mathsf{imm}}}{\to} \cdots \overset{\mathsf{mo}|_{\mathsf{imm}}}{\to} w_{(m,n_m)}}_{T_m}$$

Were this not the case and we had $w_1 \overset{\mathsf{mo}}{\to} w \overset{\mathsf{mo}}{\to} w_2$ such that $w_1, w_2 \in T_i$ and $w \in T_j \neq T_i$, we would consequently have $w_1 \overset{\mathsf{mo}_T}{\to} w \overset{\mathsf{mo}_T}{\to} w_1$, contradicting the assumption that $\Gamma$ is consistent. Given the above order, let us then define $\Gamma.\mathsf{MO}_{\mathtt{x}} = [T_1 \cdots T_m]$. We write $\Gamma.\mathsf{MO}_{\mathtt{x}}|_i$ for the i$^{\text{th}}$ item of $\Gamma.\mathsf{MO}_{\mathtt{x}}$. As we describe shortly, we use $\Gamma.\mathsf{MO}_{\mathtt{x}}$ to determine the order of lock events.

Note that the execution trace for each transaction $T_i \in \Gamma.T/\mathsf{st}$ is of the form $\theta'_i = B_i \overset{\mathsf{po}}{\to} Ts'_i \overset{\mathsf{po}}{\to} E_i$, where $B_i$ is a transaction-begin ($\mathtt{B}$) event, $E_i$ is a transaction-end ($\mathtt{E}$) event, and $Ts'_i = t'_1 \overset{\mathsf{po}}{\to} \cdots \overset{\mathsf{po}}{\to} t'_n$ for some $n$, where each $t'_j$ is either a read or a write event. As such, we have $\Gamma.E = \Gamma.T = \bigcup_{T_i \in \Gamma.T/\mathsf{st}} T_i = \theta'_i.E$.

For each trace $\theta'_i$ of the specification, we construct a corresponding trace of our implementation $\theta_i$ as follows. Let $\mathtt{RS}_{T_i} = \{(\mathtt{x}_1, r_1) \cdots (\mathtt{x}_p, r_p)\}$ and $\mathtt{WS}_{T_i} = \{\mathtt{y}_1 \cdots \mathtt{y}_q\}$. We then construct $\theta_i = Ls_i \overset{\mathsf{po}}{\to} S_i \overset{\mathsf{po}}{\to} Ts_i \overset{\mathsf{po}}{\to} Us_i$, where

- $Ls_i = L_i^{\mathtt{y}_1} \overset{\mathsf{po}}{\to} \cdots \overset{\mathsf{po}}{\to} L_i^{\mathtt{y}_q}$ and $Us_i = U_i^{\mathtt{y}_1} \overset{\mathsf{po}}{\to} \cdots \overset{\mathsf{po}}{\to} U_i^{\mathtt{y}_q}$ denote the sequence of events acquiring and releasing the version locks, respectively. Each $L_i^{\mathtt{y}_j}$ and $U_i^{\mathtt{y}_j}$ are defined as follows, the first event $L_i^{\mathtt{y}_1}$ has the same identifier as that of $B_i$, the last event $U_i^{\mathtt{y}_q}$ has the same identifier as that of $E_i$, and the identifiers of the remaining events are picked fresh:

$$L_i^{\mathtt{y}_j} = (\mathtt{U}, \mathtt{vy}_j, 2a, 2a{+}1) \quad U_i^{\mathtt{y}_j} = (\mathtt{W}, \mathtt{vy}_j, 2a{+}2) \text{ where } \mathsf{MO}_{\mathtt{y}_j}\Big|_a = T_i$$

  We then define the $\mathsf{mo}$ relation for version locks such that if transaction $T_i$ writes to $\mathtt{y}$ immediately after $T_j$ (i.e. $T_i$ is $\mathsf{MO}_{\mathtt{y}}$-ordered immediately after $T_j$), then $T_i$ acquires the $\mathtt{vy}$ version lock immediately after $T_j$ has released it. On the other hand, if $T_i$ is the first transaction to write to $\mathtt{y}$, then it acquires $\mathtt{vy}$ immediately after the event initialising the value of $\mathtt{vy}$, written $init_{\mathtt{vy}}$. Moreover, each $\mathtt{vy}$ release event of $T_i$ is $\mathsf{mo}$-ordered immediately after the corresponding $\mathtt{vy}$ acquisition event in $T_i$:

$$\mathsf{IMO}_i \triangleq \bigcup_{\mathtt{y} \in \mathtt{WS}_{T_i}} \left\{ \begin{pmatrix} (L_i^{\mathtt{y}}, U_i^{\mathtt{y}}), \\ (w, L_i^{\mathtt{y}}) \end{pmatrix} \middle| \begin{array}{l} (\Gamma.\mathsf{MO}_{\mathtt{x}}|_0 = T_i \Rightarrow w = init_{\mathtt{vy}}) \wedge \\ (\exists T_j, a > 0.\ \ \Gamma.\mathsf{MO}_{\mathtt{y}}\big|_a = T_i \wedge \Gamma.\mathsf{MO}_{\mathtt{y}}\big|_{a-1} = T_j \\ \qquad \Rightarrow w = U_j^{\mathtt{y}}) \end{array} \right\}$$

  This partial $\mathsf{mo}$ order on lock events of $T_i$ also determines the $\mathsf{rf}$ relation for its lock acquisition events: $\mathsf{IRF}_i^1 \triangleq \bigcup_{\mathtt{y} \in \mathtt{WS}_{T_i}} \left\{ (w, L_i^{\mathtt{y}}) \,\middle|\, (w, L_i^{\mathtt{y}}) \in \mathsf{IMO}_i \right\}$.
- $S_i = tr_i^{\mathtt{x}_1} \overset{\mathsf{po}}{\to} \cdots \overset{\mathsf{po}}{\to} tr_i^{\mathtt{x}_p} \overset{\mathsf{po}}{\to} vr_i^{\mathtt{x}_1} \overset{\mathsf{po}}{\to} \cdots \overset{\mathsf{po}}{\to} vr_i^{\mathtt{x}_p}$ denotes the sequence of events obtaining a tentative snapshot ($tr_i^{\mathtt{x}_j}$) and subsequently validating it ($vr_i^{\mathtt{x}_j}$). Each $tr_i^{\mathtt{x}_j}$ sequence is defined as $ir_i^{\mathtt{x}_j} \overset{\mathsf{po}}{\to} r_i^{\mathtt{x}_j} \overset{\mathsf{po}}{\to} s_i^{\mathtt{x}_j}$ (reading the version lock $\mathtt{vx}_j$, reading $\mathtt{x}_j$ and recoding it in $\mathtt{s}$), with $ir_i^{\mathtt{x}_j}$, $r_i^{\mathtt{x}_j}$, $s_i^{\mathtt{x}_j}$ and $vr_i^{\mathtt{x}_j}$ events defined as follows (with fresh identifiers). We then define the $\mathsf{rf}$ relation for each of these read events in $S_i$. For each $(\mathtt{x}, r) \in \mathtt{RS}_{T_i}$, when $r$ (i.e. the read event in the specification class $T_i$ that reads the value of $\mathtt{x}$) reads from an event $w$ in the specification graph ($(w, r) \in \Gamma.\mathsf{rf}$), we add $(w, r_i^{\mathtt{x}})$ to the $\mathsf{rf}$ relation of $G$ (the first line of $\mathsf{IRF}_i^2$ below). For version locks, if transaction $T_i$ also writes to $\mathtt{x}_j$, then $ir_i^{\mathtt{x}_j}$ and $vr_i^{\mathtt{x}_j}$ events (reading and validating the

value of version lock $\mathtt{vx}_j$), read from the lock event in $T_i$ that acquired $\mathtt{vx}_j$, namely $L_i^{\mathtt{x}_j}$. On the other hand, if transaction $T_i$ does not write to $\mathtt{x}_j$ and it reads the value of $\mathtt{x}_j$ written by $T_j$, then $ir_i^{\mathtt{x}_j}$ and $vr_i^{\mathtt{x}_j}$ read the value written to $\mathtt{vx}_j$ by $T_j$ when releasing it ($U_j^{\mathtt{x}}$). Lastly, if $T_i$ does not write to $\mathtt{x}_j$ and it reads the value of $\mathtt{x}_j$ written by the initial write, $init_{\mathtt{x}}$, then $ir_i^{\mathtt{x}_j}$ and $vr_i^{\mathtt{x}_j}$ read the value written to $\mathtt{vx}_j$ by the initial write to $\mathtt{vx}$, $init_{\mathtt{vx}}$.

$$\mathsf{IRF}_i^2 \triangleq \bigcup_{(\mathtt{x},r)\in \mathtt{RS}_{T_i}} \left\{ \begin{array}{l|l} (w, r_i^{\mathtt{x}}), & (w,r)\in \mathit{\Gamma}.\mathsf{rf} \\ (w', ir_i^{\mathtt{x}}), & \wedge\, (\mathtt{x}\in \mathtt{WS}_{T_i} \Rightarrow w'{=}L_i^{\mathtt{x}}) \\ (w', vr_i^{\mathtt{x}}) & \wedge\, (\mathtt{x}\notin \mathtt{WS}_{T_i} \wedge \exists T_j.\ w\in T_j \Rightarrow w'{=}U_j^{\mathtt{x}}) \\ & \wedge\, (\mathtt{x}\notin \mathtt{WS}_{T_i} \wedge w{=}init_{\mathtt{x}} \Rightarrow w'{=}init_{\mathtt{vx}}) \end{array} \right\}$$

$$r_i^{\mathtt{x}_j}{=}(\mathtt{R},\mathtt{x}_j,v) \quad s_i^{\mathtt{x}_j}{=}(\mathtt{W},\mathtt{s}\,[\mathtt{x}_j]\,,v) \quad \text{s.t. } \exists w.\ (w,r_i^{\mathtt{x}_j})\in \mathsf{IRF}_i^2 \wedge \mathtt{val}_{\mathtt{w}}(w){=}v$$

$$ir_i^{\mathtt{x}_j}{=}vr_i^{\mathtt{x}_j}{=}(\mathtt{R},\mathtt{vx}_j,v) \quad \text{s.t. } \exists w.\ (w,ir_i^{\mathtt{x}_j})\in \mathsf{IRF}_i^2 \wedge \mathtt{val}_{\mathtt{w}}(w){=}v$$

- $Ts_i = t_1 \overset{\mathsf{po}}{\to} \cdots \overset{\mathsf{po}}{\to} t_n$ (when $Ts'_i = t'_1 \overset{\mathsf{po}}{\to} \cdots \overset{\mathsf{po}}{\to} t'_n$), with $t_j$ defined as follows:

$$t_j = (\mathtt{R},\mathtt{s}\,[\mathtt{x}]\,,v) \text{ when } t'_j = (\mathtt{R},\mathtt{x},v)$$
$$t_j = (\mathtt{W},\mathtt{x},v) \overset{\mathsf{po}|_{\mathsf{imm}}}{\to} (\mathtt{W},\mathtt{s}\,[\mathtt{x}]\,,v) \text{ when } t'_j = (\mathtt{W},\mathtt{x},v)$$

  When $t'_j$ is a read event, the $t_j$ has the same identifier as that of $t'_j$. When $t'_j$ is a write event, the first event in $t_j$ has the same identifier as that of $t_j$ and the identifier of the second event is picked fresh.

We are now in a position to construct our implementation graph. Given a consistent execution graph $\mathit{\Gamma}$ of the specification, we construct an execution graph $G = (E, \mathsf{po}, \mathsf{rf}, \mathsf{mo})$ of the implementation as follows.

- $G.E = \bigcup_{T_i\in \mathit{\Gamma}.T/\mathsf{st}} \theta_i.E$ – note that $G.E$ is an extension of $\mathit{\Gamma}.E$: $\mathit{\Gamma}.E \subseteq G.E$.
- $G.\mathsf{po}$ is defined as $\mathit{\Gamma}.\mathsf{po}$ extended by the $\mathsf{po}$ for the additional events of $G$, given by the $\theta_i$ traces defined above.
- $G.\mathsf{rf} = \bigcup_{T_i\in \mathit{\Gamma}.T/\mathsf{st}} (\mathsf{IRF}_i^1 \cup \mathsf{IRF}_i^2)$
- $G.\mathsf{mo} = \mathit{\Gamma}.\mathsf{mo} \cup \left( \bigcup_{T_i\in \mathit{\Gamma}.T/\mathsf{st}} \mathsf{IMO}_i \right)^+$

## 5 Robust Parallel Snapshot Isolation (RPSI)

In the previous section we adapted the PSI semantics in [13] to STM settings, in the *absence* of non-transactional code. However, a reasonable STM should account for mixed-mode code where shared data is accessed by both transactional and non-transactional code. To remedy this, we explore the semantics of PSI STMs in the presence of non-transactional code with *weak isolation* guarantees (see §2.1). We refer to the weakly isolated behaviour of such PSI STMs as *robust parallel snapshot isolation* (RPSI), due to its ability to provide PSI guarantees between transactions even in the presence of non-transactional code.

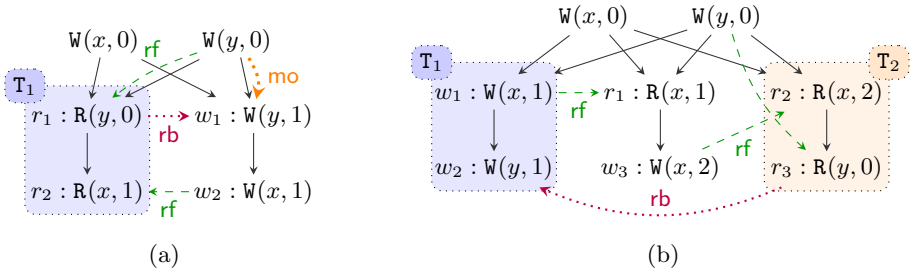


Fig. 3: RPSI-inconsistent executions due to NT-RF (a); and T-RF (b)

In §5.1 we propose the first declarative specification of RPSI STM programs. Later in §5.2 we develop a lock-based reference implementation of our RPSI specification in the RA fragment. We then demonstrate that our implementation is both sound (§5.3) and complete (§5.4) with respect to our proposed specification.

### 5.1 A Declarative Specification of RPSI STMs in RA

We formulate a declarative specification of RPSI semantics by adapting the PSI semantics presented in §4.1 to account for non-transactional accesses. As with the PSI specification in §4.1, we characterise the set of executions admitted by RPSI as graphs that lack cycles of certain shapes. More concretely, as with the PSI specification, we consider an RPSI execution graph to be *consistent* if $\mathsf{acyclic}(\mathsf{rpsi\text{-}hb}_{loc} \cup \mathsf{mo} \cup \mathsf{rb})$ holds, where rpsi-hb denotes the *'RPSI-happens-before'* relation, extended from that of PSI psi-hb, as described below.

**Definition 4 (RPSI consistency).** An RPSI execution graph $\Gamma = (E, \mathsf{po}, \mathsf{rf}, \mathsf{mo}, T)$ is consistent, written $\mathsf{rpsi\text{-}consistent}(\Gamma)$, if $\mathsf{acyclic}(\mathsf{rpsi\text{-}hb}_{loc} \cup \mathsf{mo} \cup \mathsf{rb})$ holds, where rpsi-hb denotes the *'RPSI-happens-before'* relation, defined as the smallest relation that satisfies the following conditions:

$$\mathsf{rpsi\text{-}hb}; \mathsf{rpsi\text{-}hb} \subseteq \mathsf{rpsi\text{-}hb} \qquad \text{(TRANS)}$$

$$\mathsf{po} \cup \mathsf{rf} \cup \mathsf{mo}_{\mathsf{T}} \subseteq \mathsf{rpsi\text{-}hb} \qquad \text{(PSI-HB)}$$

$$[E \setminus T]; \mathsf{rf}; \mathsf{st} \subseteq \mathsf{rpsi\text{-}hb} \qquad \text{(NT-RF)}$$

$$\mathsf{st}; ([W]; \mathsf{st}; (\mathsf{rpsi\text{-}hb} \setminus \mathsf{st}); \mathsf{st}; [R])_{loc}; \mathsf{st} \subseteq \mathsf{rpsi\text{-}hb} \qquad \text{(T-RF)}$$

The TRANS and PSI-HB ensure that rpsi-hb is transitive and that it includes po, rf and $\mathsf{mo}_{\mathsf{T}}$ as with its PSI counterpart. The NT-RF ensures that if a value written by a non-transactional write $w$ is observed (read from) by a read event $r$ in a transaction T, then its effect is observed by *all* events in T. That is, the $w$ *happens-before* all events in T and not just $r$. This allows us to rule out executions such as the one depicted in Fig. 3a, which we argue must be disallowed by RPSI.

Consider the execution graph of Fig. 3a, where transaction $\mathtt{T}_1$ is denoted by the dashed box labelled $\mathtt{T}_1$, comprising the read events $r_1$ and $r_2$. Note that as

$r_1$ and $r_2$ are transactional reads without prior writes by the transaction, they constitute a *snapshot* of the memory at the time $\mathtt{T}_1$ started. That is, the values read by $r_1$ and $r_2$ must reflect a valid snapshot of the memory at the time it was taken. As such, since we have $(w_2, r_2) \in \mathsf{rf}$, any event preceding $w_2$ by the 'happens-before' relation must also be observed by (synchronise with) $\mathtt{T}_1$. In particular, as $w_1$ happens-before $w_2$ ($(w_1, w_2) \in \mathsf{po}$), the $w_1$ write must also be observed by $\mathtt{T}_1$. The NT-RF thus ensures that a non-transactional write read from by a transaction (i.e. a snapshot read) synchronises with the entire transaction.

Recall from §4.1 that the PSI psi-hb relation includes $\mathsf{rf}_\mathsf{T}$ which has not yet been included in rpsi-hb through the first three conditions described. As we describe shortly, the T-RF is indeed a strengthening of $\mathsf{rf}_\mathsf{T}$ to account for the presence of non-transactional events. In particular, note that $\mathsf{rf}_\mathsf{T}$ is included in the left-hand side of T-RF: when rpsi-hb in $([W]; \mathsf{st}; (\mathsf{rpsi\text{-}hb} \setminus \mathsf{st}); \mathsf{st}; [R])$ is replaced with $\mathsf{rf} \subseteq \mathsf{rpsi\text{-}hb}$, the left-hand side yields $\mathsf{rf}_\mathsf{T}$. As such, in the absence of non-transactional events, the definitions of psi-hb and rpsi-hb coincide.

Recall that inclusion of $\mathsf{rf}_\mathsf{T}$ in psi-hb ensured transactional synchronisation due to causal ordering: if $\mathtt{T}_1$ writes to $\mathtt{x}$ and $\mathtt{T}_2$ later (in psi-hb order) reads $\mathtt{x}$, then $\mathtt{T}_1$ must synchronise with $\mathtt{T}_2$. This was achieved in PSI because either i) $\mathtt{T}_2$ reads $\mathtt{x}$ directly from $\mathtt{T}_1$ in which case $\mathtt{T}_1$ synchronises with $\mathtt{T}_2$ via $\mathsf{rf}_\mathsf{T}$; or ii) $\mathtt{T}_2$ reads $\mathtt{x}$ from another later (mo-ordered) transactional write in $\mathtt{T}_3$, in which case $\mathtt{T}_1$ synchronises with $\mathtt{T}_3$ via $\mathsf{mo}_\mathsf{T}$, $\mathtt{T}_3$ synchronises with $\mathtt{T}_2$ via $\mathsf{rf}_\mathsf{T}$, and thus $\mathtt{T}_1$ synchronises with $\mathtt{T}_2$ via $\mathsf{mo}_\mathsf{T}; \mathsf{rf}_\mathsf{T}$. How are we then to extend rpsi-hb to guarantee transactional synchronisation due to causal ordering in the presence of non-transactional events?

To justify T-RF, we present an execution graph that does not guarantee synchronisation between causally ordered transactions and is nonetheless deemed RPSI-consistent *without* the T-RF condition on rpsi-hb. We thus argue that this execution must be precluded by RPSI, justifying the need for T-RF. Consider the execution in Fig. 3b. Observe that as transaction $\mathtt{T}_1$ writes to $x$ via $w_1$, transaction $\mathtt{T}_2$ reads $x$ via $r_2$, and $(w_1, r_2) \in \mathsf{rpsi\text{-}hb}$ ($w_1 \xrightarrow{\mathsf{rf}} r_1 \xrightarrow{\mathsf{po}} w_3 \xrightarrow{\mathsf{rf}} r_2$), $\mathtt{T}_1$ is causally ordered before $\mathtt{T}_2$ and hence $\mathtt{T}_1$ must synchronise with $\mathtt{T}_2$. As such, the $r_3$ in $\mathtt{T}_2$ must observe $w_2$ in $\mathtt{T}_1$: we must have $(w_2, r_3) \in \mathsf{rpsi\text{-}hb}$, rendering the above execution RPSI-inconsistent. To enforce the rpsi-hb relation between such causally ordered transactions with intermediate non-transactional events, T-RF stipulates that if a transaction $\mathtt{T}_1$ writes to a location (e.g. $x$ via $w_1$ above), another transaction $\mathtt{T}_2$ reads from the same location ($r_2$), and the two events are related by 'RPSI-happens-before' ($(w_1, r_2) \in \mathsf{rpsi\text{-}hb}$), then $\mathtt{T}_1$ must synchronise with $\mathtt{T}_2$. That is, all events in $\mathtt{T}_1$ must 'RPSI-happen-before' those in $\mathtt{T}_2$. Effectively, this allows us to transitively close the causal ordering between transactions, spanning transactional and non-transactional events in between.

### 5.2 A Lock-based RPSI Implementation in RA

We present a lock-based reference implementation of RPSI in the RA fragement (Fig. 4) by using sequence locks [17,22,31,12]. Our implementation is both sound and complete with respect to our declarative RPSI specification in §5.1.

```
begin;  for(x in WS) {  lock vx; }
  T;    snapshot(RS); T';
end     for(x in WS) { unlock vx; }
```

where `snapshot(RS)` $\triangleq$ `do {`

```
    for(x in RS) {
      a := vx;
      if (is-odd(a) && !(x in WS)) { continue; }
      if (!(x in WS)) { v[x]:= a }
      s[x] := x;
    } b := true;
    for(x in RS) {
      if (!b) { break; }
      a := x; v := vx;
      b := (v[x] == v && s[x] == a);
  } } while (!b)
```

and `lock vx`, `T'` and `unlock vx` are as defined in Fig. 2.

Fig. 4: Lock-based RPSI implementation of `T` given `RS` and `WS`

As before, in order to avoid taking a snapshot of the entire memory, in our implementation we assume that a transaction `T` is supplied with its read set, `RS`, and its write set, `WS`. The RPSI implementation in Fig. 4 is rather similar to that of PSI implementation in Fig. 2. In particular, as with its PSI counterpart, the RPSI implementation of `T` acquires the version locks on its write set and subsequently obtains a snapshot `s` of its read set. The `T` is then executed locally as before, where each read consults `s` and each write updates the memory in-place and duly updates `s` to ensure correct lookup for future reads. Once the execution of `T` is concluded, the version locks on the write set are released.

The main difference between the RPSI implementation and that of PSI is in how they obtain a snapshot `s` when calling `snapshot(RS)`. More concretely, the initial phase of `snapshot(RS)` recording a *tentative* snapshot in `s` (i.e. the first `for` loop) is identical for both implementations. The difference of the two lies in the *validation* phase of `snapshot(RS)` (i.e. the second `for` loop). As before, in order to ensure that no intermediate *transactional* writes have intervened since `s` was recorded, for each location `x` in `RS`, the validation phase revisits `vx`, inspecting whether its value has changed from that recorded in `v[x]`. If this is the case, the snapshot is deemed invalid and the process is restarted. However, checking against intermediate transactional writes alone is not sufficient as it does not preclude the intervention of *non-transactional* writes. This is because unlike transactional writes, non-transactional writes do not update the version locks and as such their updates may go unnoticed. In order to rule out the possibility

of intermediate non-transactional writes, for each location `x` the implementation checks the value of `x` against that recorded in `s[x]`. If the values do not agree, an intermediate non-transactional write has been detected: the snapshot fails validation and the process is restarted. Otherwise, the snapshot is successfully validated and returned in `s`. Observe that checking the value of `x` against `s[x]` does not entirely preclude the presence of non-transactional writes, due to the ABA problem (where the value of `x` may have changed from $v$ to $v'$ and back to $v$). To rule out such scenarios and guarantee the absence of intermediate writes, we assume that each *non-transactional* write to a location `x` *writes a unique value*, distinct from those of other writes to `x`, both transactional and non-transactional.

Note that this latter stipulation does not prevent two *transactions* to write the same value to a location `x`. As such, in the absence of non-transactional writes, our RPSI implementation is equivalent to that of PSI in §4.2.

### 5.3 Implementation Soundness

The RPSI implementation in Fig. 4 is *sound*: for each consistent implementation graph $G$, a corresponding specification graph $\Gamma$ can be constructed such that $\mathsf{rpsi\text{-}consistent}(\Gamma)$ holds. In what follows we state our soundness theorem and briefly describe our construction of consistent specification graphs. We refer the reader to the technical appendix for the full soundness proof.

**Theorem 3 (Soundness).** *For all RA-consistent implementation graphs $G$ (with unique non-transactional written values) of the implementation in Fig. 4, there exists an RPSI-consistent specification graph $\Gamma$ of the corresponding transactional program with the same program outcome.*

**Constructing Consistent Specification Graphs** Constructing an RPSI-consistent specification graph from the implementation graph is similar to the corresponding PSI construction described in §4.3. More concretely, the events associated with non-transactional events remain unchanged and are simply added to the specification graph. On the other hand, the events associated with transactional events are adapted in a similar way to those of PSI in §4.3. In particular, observe that given an execution of the RPSI implementation with $t$ transactions, as with the PSI implementation, the trace of each transaction $i \in \{1 \cdots t\}$ is of the form $\theta_i = Ls_i \xrightarrow{\mathsf{po}} FS_i \xrightarrow{\mathsf{po}} S_i \xrightarrow{\mathsf{po}} Ts_i \xrightarrow{\mathsf{po}} Us_i$, with $Ls_i$, $FS_i$, $S_i$, $Ts_i$ and $Us_i$ denoting analogous sequences of events to those of PSI. The difference between an RPSI trace $\theta_i$ and a PSI one is in the $FS_i$ and $S_i$ sequences, obtaining the snapshot. In particular, the validation phases of $FS_i$ and $S_i$ in RPSI include an additional read for each location to rule out intermediate non-transactional writes. As in the PSI construction, for each transactional trace $\theta_i$ of our implementation, we construct a corresponding trace of the specification as $\theta'_i = B_i \xrightarrow{\mathsf{po}} Ts'_i \xrightarrow{\mathsf{po}} E_i$, with $B_i$, $E_i$ and $Ts'_i$ as defined in §4.3.

Given a consistent RPSI implementation graph $G = (E, \mathsf{po}, \mathsf{rf}, \mathsf{mo})$, let $G.NT \triangleq G.E \setminus \bigcup_{i \in \{1 \cdots t\}} \theta.E$ denote the non-transactional events of $G$. We construct a consistent RPSI specification graph $\Gamma = (E, \mathsf{po}, \mathsf{rf}, \mathsf{mo}, T)$ such that:

- $\Gamma.E \triangleq G.NT \cup \bigcup_{i \in \{1 \cdots t\}} \theta'_i.E$ – the $\Gamma.E$ events comprise the non-transactional events in $G$ and the events in each transactional trace $\theta'_i$ of the specification;
- $\Gamma.\mathsf{po} \triangleq G.\mathsf{po}|_{\Gamma.E}$ – the $\Gamma.\mathsf{po}$ is that of $G.\mathsf{po}$ restricted to the events in $\Gamma.E$;
- $\Gamma.\mathsf{rf} \triangleq \bigcup_{i \in \{1 \cdots t\}} \mathsf{RF}_i \cup G.\mathsf{rf}; [G.NT]$ – the $\Gamma.\mathsf{rf}$ is the union of $\mathsf{RF}_i$ relations for transactional reads as defined in §4.3, together with the $G.\mathsf{rf}$ relation for non-transactional reads;
- $\Gamma.\mathsf{mo} \triangleq G.\mathsf{mo}|_{\Gamma.E}$ – the $\Gamma.\mathsf{mo}$ is that of $G.\mathsf{mo}$ restricted to the events in $\Gamma.E$;
- $\Gamma.T \triangleq \bigcup_{i \in \{1 \cdots t\}} \theta'_i.E$, where for each $e \in \theta'_i.E$, we define $\mathtt{tx}(e) = i$.

We refer the reader to the technical appendix for the full proof demonstrating that the above construction of $\Gamma$ yields a consistent specification graph.

### 5.4 Implementation Completeness

The RPSI implementation in Fig. 4 is *complete*: for each consistent specification graph $\Gamma$ a corresponding implementation graph $G$ can be constructed such that $\mathsf{consistent}(G)$ holds. We next state our completeness theorem and describe our construction of consistent implementation graphs. We refer the reader to the technical appendix for the full completeness proof.

**Theorem 4 (Completeness).** *For all RPSI-consistent specification graphs $\Gamma$ of a program, there exists an RA-consistent execution graph $G$ of the implementation in Fig. 4 that has the same program outcome.*

**Constructing Consistent Implementation Graphs** In order to construct an execution graph of the implementation $G$ from the specification $\Gamma$, we follow similar steps as those in the corresponding PSI construction in §4.4. More concretely, the events associated with non-transactional events are unchanged and simply added to the implementation graph. For transactional events, given each trace $\theta'_i$ of a transaction in the specification, as before we construct an analogous trace of the implementation by inserting the appropriate events for acquiring and inspecting the version locks, as well as obtaining a snapshot. For each transaction class $T_i \in T/\mathsf{st}$, we first determine its read and write sets as before and subsequently decide the order in which the version locks are acquired and inspected. This then enables us to construct the 'reads-from' and 'modification-order' relations for the events associated with version locks.

Given a consistent execution graph of the specification $\Gamma = (E, \mathsf{po}, \mathsf{rf}, \mathsf{mo}, T)$, and a transaction class $T_i \in \Gamma.T/\mathsf{st}$, we define $\mathtt{WS}_{T_i}$ and $\mathtt{RS}_{T_i}$ as described in §4.4. Determining the ordering of lock events hinges on a similar observation as that in the PSI construction. Given a consistent execution graph of the specification $\Gamma = (E, \mathsf{po}, \mathsf{rf}, \mathsf{mo}, T)$, let for each location $\mathtt{x}$ the total order $\mathsf{mo}$ be given as: $w_1 \overset{\mathsf{mo}|_{\mathrm{imm}}}{\rightarrow} \cdots \overset{\mathsf{mo}|_{\mathrm{imm}}}{\rightarrow} w_{n_\mathtt{x}}$. This order can be broken into adjacent segments where the events of each segment are *either* non-transactional writes *or* belong to the *same* transaction. That is, given the transaction classes $\Gamma.T/\mathsf{st}$, the order above

is of the following form where $T_1, \cdots, T_m \in \Gamma.T/\mathsf{st}$ and for each such $T_i$ we have $\mathtt{x} \in \mathtt{WS}_{T_i}$ and $w_{(i,1)} \cdots w_{(i,n_i)} \in T_i$:

$$\underbrace{w_{(1,1)} \overset{\mathsf{mo}|_{\mathrm{imm}}}{\to} \cdots \overset{\mathsf{mo}|_{\mathrm{imm}}}{\to} w_{(1,n_1)}}_{\Gamma.NT \cup T_1} \overset{\mathsf{mo}|_{\mathrm{imm}}}{\to} \cdots \overset{\mathsf{mo}|_{\mathrm{imm}}}{\to} \underbrace{w_{(m,1)} \overset{\mathsf{mo}|_{\mathrm{imm}}}{\to} \cdots \overset{\mathsf{mo}|_{\mathrm{imm}}}{\to} w_{(m,n_m)}}_{\Gamma.NT \cup T_m}$$

Were this not the case and we had $w_1 \overset{\mathsf{mo}}{\to} w \overset{\mathsf{mo}}{\to} w_2$ such that $w_1, w_2 \in T_i$ and $w \in T_j \neq T_i$, we would consequently have $w_1 \overset{\mathsf{mo}_T}{\to} w \overset{\mathsf{mo}_T}{\to} w_1$, contradicting the assumption that $\Gamma$ is consistent. We thus define $\Gamma.\mathsf{MO}_{\mathtt{x}} = [T_1 \cdots T_m]$.

Note that each transactional execution trace of the specification is of the form $\theta'_i = B_i \overset{\mathsf{po}}{\to} Ts'_i \overset{\mathsf{po}}{\to} E_i$, with $B_i$, $E_i$ and $Ts'_i$ as described in §4.4. For each such $\theta'_i$, we construct a corresponding trace of our implementation as $\theta_i = Ls_i \overset{\mathsf{po}}{\to} S_i \overset{\mathsf{po}}{\to} Ts_i \overset{\mathsf{po}}{\to} Us_i$, where $Ls_i$, $Ts_i$ and $Us_i$ are as defined in §4.4, and $S_i = tr_i^{\mathtt{x}_1} \overset{\mathsf{po}}{\to} \cdots \overset{\mathsf{po}}{\to} tr_i^{\mathtt{x}_p} \overset{\mathsf{po}}{\to} vr_i^{\mathtt{x}_1} \overset{\mathsf{po}}{\to} \cdots \overset{\mathsf{po}}{\to} vr_i^{\mathtt{x}_p}$ denotes the sequence of events obtaining a tentative snapshot ($tr_i^{\mathtt{x}_j}$) and subsequently validating it ($vr_i^{\mathtt{x}_j}$). Each $tr_i^{\mathtt{x}_j}$ sequence is of the form $ivr_i^{\mathtt{x}_j} \overset{\mathsf{po}}{\to} ir_i^{\mathtt{x}_j} \overset{\mathsf{po}}{\to} s_i^{\mathtt{x}_j}$, with $ivr_i^{\mathtt{x}_j}$, $ir_i^{\mathtt{x}_j}$ and $s_i^{\mathtt{x}_j}$ defined below (with fresh identifiers). Similarly, each $vr_i^{\mathtt{x}_j}$ sequence is of the form $fr_i^{\mathtt{x}_j} \overset{\mathsf{po}}{\to} fvr_i^{\mathtt{x}_j}$, with $fr_i^{\mathtt{x}_j}$ and $fvr_i^{\mathtt{x}_j}$ defined as follows (with fresh identifiers). We then define the $\mathsf{rf}$ relation for each of these read events in $S_i$ in a similar way.

For each $(\mathtt{x}, r) \in \mathtt{RS}_{T_i}$, when $r$ (the event in the specification class $T_i$ that reads the value of $\mathtt{x}$) reads from $w$ in the specification graph ($(w, r) \in \Gamma.\mathsf{rf}$), we add $(w, ir_i^{\mathtt{x}})$ and $(w, fr_i^{\mathtt{x}})$ to the $\mathsf{rf}$ of $G$ (the first line of $\mathsf{IRF}_i^2$ below). For version locks, as before if transaction $T_i$ also writes to $\mathtt{x}_j$, then $ivr_i^{\mathtt{x}_j}$ and $fvr_i^{\mathtt{x}_j}$ events (reading and validating $\mathtt{vx}_j$), read from the lock event in $T_i$ that acquired $\mathtt{vx}_j$, namely $L_i^{\mathtt{x}_j}$. Similarly, if $T_i$ does not write to $\mathtt{x}_j$ and it reads the value of $\mathtt{x}_j$ written by the initial write, $init_{\mathtt{x}}$, then $ivr_i^{\mathtt{x}_j}$ and $fvr_i^{\mathtt{x}_j}$ read the value written to $\mathtt{vx}_j$ by the initial write to $\mathtt{vx}$, $init_{\mathtt{vx}}$. Lastly, if transaction $T_i$ does not write to $\mathtt{x}_j$ and it reads $\mathtt{x}_j$ from a write other than $init_{\mathtt{x}}$, then $ir_i^{\mathtt{x}_j}$ and $vr_i^{\mathtt{x}_j}$ read from the unlock event of a transaction $T_j$ (i.e. $U_j^{\mathtt{x}}$), who has $\mathtt{x}$ in its write set and whose write to $\mathtt{x}$, $w_{\mathtt{x}}$, maximally 'RPSI-happens-before' $r$. That is, for all other such writes that 'RPSI-happen-before' $r$, then $w_{\mathtt{x}}$ 'RPSI-happens-after' them.

$$\mathsf{IRF}_i^2 \triangleq \bigcup_{(\mathtt{x},r) \in \mathtt{RS}_{T_i}} \left\{ \begin{array}{l|l} (w, ir_i^{\mathtt{x}}), \\ (w, fr_i^{\mathtt{x}}), \\ (w', ivr_i^{\mathtt{x}}), \\ (w', fvr_i^{\mathtt{x}}) \end{array} \middle| \begin{array}{l} (w, r) \in \Gamma.\mathsf{rf} \wedge (\mathtt{x} \in \mathtt{WS}_{T_i} \Rightarrow w' = L_i^{\mathtt{x}}) \\ \wedge\, (\mathtt{x} \notin \mathtt{WS}_{T_i} \wedge w = init_{\mathtt{x}} \Rightarrow w' = init_{\mathtt{vx}}) \\ \wedge\, (\mathtt{x} \notin \mathtt{WS}_{T_i} \wedge w \neq init_{\mathtt{x}} \Rightarrow \\ \quad \exists w_{\mathtt{x}}, T_j.\ w_{\mathtt{x}} \in T_j \cap W_{\mathtt{x}} \wedge w_{\mathtt{x}} \overset{\mathsf{rpsi\text{-}hb}}{\to} r \wedge w' = U_j^{\mathtt{x}} \\ \quad \wedge [\forall w'_{\mathtt{x}}, T_k.\ w'_{\mathtt{x}} \in T_k \cap W_{\mathtt{x}} \wedge w'_{\mathtt{x}} \overset{\mathsf{rpsi\text{-}hb}}{\to} r \Rightarrow w'_{\mathtt{x}} \overset{\mathsf{rpsi\text{-}hb}}{\to} w_{\mathtt{x}}]) \end{array} \right\}$$

$ir_i^{\mathtt{x}_j} = fr_i^{\mathtt{x}_j} = (\mathtt{R}, \mathtt{x}_j, v) \quad s_i^{\mathtt{x}_j} = (\mathtt{W}, \mathtt{s}[\mathtt{x}_j], v) \quad$ s.t. $\exists w.\ (w, ir_i^{\mathtt{x}_j}) \in \mathsf{IRF}_i^2 \wedge \mathtt{val}_{\mathtt{w}}(w) = v$

$ivr_i^{\mathtt{x}_j} = fvr_i^{\mathtt{x}_j} = (\mathtt{R}, \mathtt{vx}_j, v)$ s.t. $\exists w.\ (w, ivr_i^{\mathtt{x}_j}) \in \mathsf{IRF}_i^2 \wedge \mathtt{val}_{\mathtt{w}}(w) = v$

We are now in a position to construct our implementation graph. Given a consistent execution graph $\Gamma$ of the specification, we construct an execution graph of the implementation, $G = (E, \mathsf{po}, \mathsf{rf}, \mathsf{mo})$, such that:

- $G.E = \bigcup_{T_i \in \Gamma.T/\mathsf{st}} \theta_i.E \cup \Gamma.NT$;

- $G.\mathsf{po}$ is defined as $\Gamma.\mathsf{po}$ extended by the $\mathsf{po}$ for the additional events of $G$, given by the $\theta_i$ traces defined above;
- $G.\mathsf{rf} = \bigcup_{T_i \in \Gamma.T/\mathsf{st}} (\mathsf{IRF}_i^1 \cup \mathsf{IRF}_i^2)$, with $\mathsf{IRF}_i^1$ as in §4.4 and $\mathsf{IRF}_i^2$ defined above;
- $G.\mathsf{mo} = \Gamma.\mathsf{mo} \cup \left( \bigcup_{T_i \in \Gamma.T/\mathsf{st}} \mathsf{IMO}_i \right)^+$, with $\mathsf{IMO}_i$ as defined in §4.4.

## 6 Conclusions and Future Work

We studied PSI, for the first time to our knowledge, as a consistency model for STMs as it has several advantages over other consistency models, thanks to its performance and monotonic behaviour. We addressed two significant drawbacks of PSI which prevent its widespread adoption. First, the absence of a simple lock-based reference implementation to allow the programmers to readily understand and reason about PSI programs. To address this, we developed a lock-based reference implementation of PSI in the RA fragment of C11 (using sequence locks), that is both sound and complete with respect to its declarative specification. Second, the absence of a formal PSI model in the presence of mixed-mode accesses. To this end, we formulated a declarative specification of RPSI (robust PSI) accounting for both transactional and non-transactional accesses. Our RPSI specification is an extension of PSI in that in the absence of non-transactional accesses it coincides with PSI. To provide a more intuitive account of RPSI, we developed a simple lock-based RPSI reference implementation by adjusting our PSI implementation. We established the soundness and completeness of our RPSI implementation against its declarative specification.

As directions of future work, we plan to build on top of the work presented here in three ways. First, we plan to explore possible lock-based reference implementations for PSI and RPSI in the context of other weak memory models, such as the full C11 memory models [8]. Second, we plan to study other weak transactional consistency models, such as SI [9], ALA (asymmetric lock atomicity), ELA (encounter-time lock atomicity) [27], and those of ANSI SQL, including RU (read-uncommitted), RC (read-committed) and RR (repeatable reads), in the STM context. We aim to investigate possible lock-based reference implementations for these models that would allow the programmers to understand and reason about STM programs with such weak guarantees. Third, taking advantage of the operational models provided by our simple lock-based implementations (those presented in this article as well as those in future work), we plan to develop reasoning techniques that would allow us to verify properties of STM programs. This can be achieved by either extending existing program logics for weak memory, or developing new program logics for currently unsupported models. In particular, we can reason about the PSI models presented here by developing custom proof rules in the existing program logics for RA such as [21,38].

## References


1. The Clojure Language: Refs and Transactions, http://clojure.org/refs
2. Haskell STM, http://hackage.haskell.org/package/stm-2.2.0.1/docs/Control-Concurrent-STM.html
3. Software transactional memory (Scala), https://doc.akka.io/docs/akka/1.2/scala/stm.html
4. Generalized isolation level definitions. In: Proceedings of the 16th International Conference on Data Engineering (2000)
5. Technical specification for C++ extensions for transactional memory (2015), http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4514.pdf
6. Adya, A.: Weak consistency: A generalized theory and optimistic implementations for distributed transactions. Ph.D. thesis, MIT (1999)
7. Alglave, J., Maranget, L., Tautschnig, M.: Herding cats: Modelling, simulation, testing, and data mining for weak memory. ACM Trans. Program. Lang. Syst. 36(2), 7:1–7:74
8. Batty, M., Owens, S., Sarkar, S., Sewell, P., Weber, T.: Mathematizing C++ concurrency. In: Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 55–66 (2011)
9. Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E., O'Neil, P.: A critique of ANSI SQL isolation levels. In: Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data. pp. 1–10 (1995)
10. Bieniusa, A., Fuhrmann, T.: Consistency in hindsight: A fully decentralized STM algorithm. In: Proceedings of the 2010 IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010. pp. 1 – 12 (2010)
11. Blundell, C., C. Lewis, E., M. K. Martin, M.: Deconstructing transactions: The subtleties of atomicity. In: 4th Annual Workshop on Duplicating, Deconstructing, and Debunking (2005)
12. Boehm, H.J.: Can seqlocks get along with programming language memory models? In: Proceedings of the 2012 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness. pp. 12–20 (2012)
13. Cerone, A., Gotsman, A.: Analysing snapshot isolation. In: Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing. pp. 55–64 (2016)
14. Cerone, A., Gotsman, A., Yang, H.: Transaction chopping for parallel snapshot isolation. In: Proceedings of the 29th International Symposium on Distributed Computing - Volume 9363. pp. 388–404 (2015)
15. Daudjee, K., Salem, K.: Lazy database replication with snapshot isolation. In: Proceedings of the 32Nd International Conference on Very Large Data Bases. pp. 715–726 (2006)
16. Harris, T., Marlow, S., Peyton-Jones, S., Herlihy, M.: Composable memory transactions. In: Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. pp. 48–60 (2005)
17. Hemminger, S.: Fast reader/writer lock for gettimeofday 2.5.30, http://lwn.net/Articles/7388/
18. Herlihy, M., Moss, J.E.B.: Transactional memory: Architectural support for lock-free data structures. In: Proceedings of the 20th Annual International Symposium on Computer Architecture. pp. 289–300 (1993)
19. Hickey, R.: The clojure programming language. In: Proceedings of the 2008 Symposium on Dynamic Languages. pp. 1:1–1:1 (2008)

20. Lahav, O., Giannarakis, N., Vafeiadis, V.: Taming release-acquire consistency. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 649–662 (2016)
21. Lahav, O., Vafeiadis, V.: Owicki-gries reasoning for weak memory models. In: Proceedings, Part II, of the 42Nd International Colloquium on Automata, Languages, and Programming - Volume 9135. pp. 311–323 (2015)
22. Lameter, C.: Effective synchronization on linux/numa systems (2005), http://www.lameter.com/gelato2005.pdf
23. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans. Comput. 28(9), 690–691 (1979)
24. Litz, H., Cheriton, D., Firoozshahian, A., Azizi, O., Stevenson, J.P.: SI-TM: Reducing transactional memory abort rates through snapshot isolation. SIGPLAN Not. pp. 383–398 (2014)
25. Litz, H., Dias, R.J., Cheriton, D.R.: Efficient correction of anomalies in snapshot isolation transactions. ACM Trans. Archit. Code Optim. pp. 65:1–65:24
26. Martin, M., Blundell, C., Lewis, E.: Subtleties of transactional memory atomicity semantics. IEEE Comput. Archit. Lett. 5(2), 17–17 (2006)
27. Menon, V., Balensiefer, S., Shpeisman, T., Adl-Tabatabai, A.R., Hudson, R.L., Saha, B., Welc, A.: Single global lock semantics in a weakly atomic STM. SIGPLAN Not. 43(5), 15–26 (2008)
28. Owens, S., Sarkar, S., Sewell, P.: A better x86 memory model: X86-tso. In: Proceedings of the 22Nd International Conference on Theorem Proving in Higher Order Logics. pp. 391–407 (2009)
29. Peng, D., Dabek, F.: Large-scale incremental processing using distributed transactions and notifications. In: Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation. pp. 251–264 (2010)
30. Pulte, C., Flur, S., Deacon, W., French, J., Sarkar, S., Sewell, P.: Simplifying ARM concurrency: Multicopy-atomic axiomatic and operational models for ARMv8 (draft) (2017)
31. Rajwar, R., Goodman, J.R.: Speculative lock elision: Enabling highly concurrent multithreaded execution. In: Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture. pp. 294–305 (2001)
32. Serrano, D., Patino-Martinez, M., Jimenez-Peris, R., Kemme, B.: Boosting database replication scalability through partial replication and 1-copy-snapshot-isolation. In: Proceedings of the 13th Pacific Rim International Symposium on Dependable Computing. pp. 290–297 (2007)
33. Sewell, P., Sarkar, S., Owens, S., Zappa Nardelli, F., Myreen, M.O.: x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. Commun. ACM 53(7), 89–97 (2010)
34. Shavit, N., Touitou, D.: Software transactional memory. In: Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing. pp. 204–213 (1995)
35. Sovran, Y., Power, R., Aguilera, M.K., Li, J.: Transactional storage for geo-replicated systems. In: Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles. pp. 385–400 (2011)
36. SPARC International, Inc., C.: The SPARC Architecture Manual: Version 8 (1992)
37. SPARC International, Inc., C.: The SPARC Architecture Manual (Version 9) (1994)
38. Vafeiadis, V., Narayan, C.: Relaxed separation logic: A program logic for c11 concurrency. In: Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications. pp. 867–884 (2013)