On the Semantics of Snapshot Isolation

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

[1] MPI-SWS, Kaiserslautern, Germany
{azalea,viktor}@mpi-sws.org
[2] Tel Aviv University, Tel Aviv, Israel
orilahav@tau.ac.il

Abstract. Snapshot isolation (SI) is a standard transactional consistency model used in databases, distributed systems and software transactional memory (STM). Its semantics is formally defined both declaratively as an acyclicity axiom, and operationally as a concurrent algorithm with memory bearing timestamps.

We develop two simpler equivalent operational definitions of SI as lock-based reference implementations that do not use timestamps. Our first locking implementation is prescient in that requires *a priori* knowledge of the data accessed by a transaction and carries out transactional writes eagerly (in-place). Our second implementation is non-prescient and performs transactional writes lazily by recording them in a local log and propagating them to memory at commit time. Whilst our first implementation is simpler and may be better suited for developing a program logic for SI transactions, our second implementation is more practical due to its non-prescience. We show that both implementations are sound and complete against the declarative SI specification and thus yield equivalent operational definitions for SI.

We further consider, for the first time *formally*, the use of SI in a context with racy non-transactional accesses, as can arise in STM implementations of SI. We introduce *robust snapshot isolation* (RSI), an adaptation of SI with similar semantics and guarantees in this mixed setting. We present a declarative specification of RSI as an acyclicity axiom and analogously develop two operational models as lock-based reference implementations (one eager, one lazy). We show that these operational models are both sound and complete against the declarative RSI model.

1 Introduction

Transactions are the *de facto* synchronisation mechanism in databases and geo-replicated distributed systems, and are thus gaining adoption in the shared-memory setting via *software transactional memory* (STM) [20,33]. In contrast to other synchronisation mechanisms, transactions readily provide atomicity, isolation, and consistency guarantees for sequences of operations, allowing programmers to focus on the high-level design of their systems.

However, providing these guarantees comes at a significant cost. As such, various transactional consistency models in the literature trade off consistency

C. Enea and R. Piskac (Eds.): VMCAI 2019, LNCS 11388, pp. 1–23, 2019.
https://doi.org/10.1007/978-3-030-11245-5_1

guarantees for better performance. At nearly the one end of the spectrum, we have *serialisability* [28], which requires transactions to appear to have been executed in some total order. Serialisability provides strong guarantees, but is widely considered too expensive to implement. The main problem is that two conflicting transactions (e.g. one reading from and one updating the same datum) cannot both execute and commit in parallel.

Consequently, most major databases, both centralised (e.g. Oracle and MS SQL Server) and distributed [15,29,32], have opted for a slightly weaker model called *snapshot isolation* (SI) [7] as their default consistency model. SI has much better performance than serialisability by allowing conflicting transactions to execute concurrently and commit successfully as long as they do not have a write-write conflict. This in effect allows reads of SI transactions to read from an earlier memory snapshot than the one affected by their writes, and permits the *write skew anomaly* [11] depicted in Fig. 1. Besides this anomaly, however, SI is essentially the same as serialisability: Cerone et al. [11] provide a widely applicable condition under which SI and serialisability coincide for a given set of transactions. For these reasons, SI has also started gaining adoption in the generic programming language setting via STM implementations [1,8,16,25,26] that provide SI semantics for their transactions.

The formal study of SI, however, has so far not accounted for the more general STM setting in which both transactions and uninstrumented non-transactional code can access the same memory locations. While there exist two equivalent definitions of SI—one declarative in terms of an acyclicity constraint [10,11] and one operational in terms of an optimistic multi-version concurrency control algorithm [7]—neither definition supports *mixed-mode* (i.e. both transactional and non-transactional) accesses to the same locations. Extending the definitions to do so is difficult for two reasons: (1) the operational definition attaches a timestamp to every memory location, which heavily relies on the absence of non-transactional accesses; and (2) there are subtle interactions between the transactional implementation and the weak memory model underlying the non-transactional accesses.

In this article, we address these limitations of SI. We develop two simple lock-based reference implementations for SI that do not use timestamps. Our first implementation is *prescient* [19] in that it requires *a priori* knowledge of the data accessed by a transaction, and performs transactional writes *eagerly* (in-place). Our second implementation is non-prescient and carries out transactional writes *lazily* by first recording them in a local log and subsequently propagating them to memory at commit time. Our first implementation is simpler and may be better suited for understanding and developing a program logic for SI transactions, whilst our second implementation is more practical due to its non-prescience. We show that both implementations are sound and complete against the declarative SI specification and thus yield equivalent operational definitions for SI.

We then extend both our eager and lazy implementations to make them robust under uninstrumented non-transactional accesses, and characterise declaratively the semantics we obtain. We call this extended model *robust snapshot* *isolation* (RSI) and show that it gives reasonable semantics with mixed-mode accesses.

To provide SI semantics, instead of timestamps, our implementations use *multiple-readers-single-writer* (MRSW) locks. They acquire locks in reader mode to take a snapshot of the memory locations accessed by a transaction and then promote the relevant locks to writer mode to enforce an ordering on transactions with write-write conflicts. As we discuss in Sect. 4, the equivalence of the RSI implementation and its declarative characterisation depends heavily upon the axiomatisation of MRSW locks: here, we opted for the weakest possible axiomatisation that does not order any concurrent reader lock operations and present an MRSW lock implementation that achieves this.

Outline. In Sect. 2 we present an overview of our contributions by describing our reference implementations for both SI and RSI. In Sect. 3 we define the declarative framework for specifying STM programs. In Sect. 4 we present the declarative SI specification against which we demonstrate the soundness and completeness of our SI implementations. In Sect. 5 we formulate a declarative specification for RSI and demonstrate the soundness and completeness of our RSI implementations. We discuss related and future work in Sect. 6.[1]

2 Background and Main Ideas

As noted earlier, the key challenge in specifying STM transactions lies in accounting for the interactions between mixed-mode accesses to the same data. One simple approach is to treat each non-transactional access as a singleton mini-transaction and to provide *strong isolation* [9,27], i.e. full isolation between transactional and non-transactional code. This, however, requires *instrumenting* non-transactional accesses to adhere to same access policies as transactional ones (e.g. acquiring the necessary locks), which incurs a substantial performance penalty for non-transactional code. A more practical approach is to enforce isolation only amongst transactional accesses, an approach known as *weak isolation* [9,27], adopted by the relaxed transactions of C++ [2].

As our focus is on STMs with SI guarantees, instrumenting non-transactional accesses is not feasible. In particular, as we expect many more non-transactional accesses than transactional ones, we do not want to incur any performance degradation on non-transactional code when executed in parallel with transactional code. As such, we opt for an STM with SI guarantees under *weak isolation*. Under weak isolation, however, transactions with explicit abort instructions are problematic as their intermediate state may be observed by non-transactional code. As such, weakly isolated STMs (e.g. C++ relaxed transactions [2]) often forbid explicit aborts altogether. Throughout our development we thus make two simplifying assumptions: (1) transactions are not nested; and (2) there are no explicit abort instructions, following the example of weakly isolated relaxed

[1] A full version of this article is available at [31].

T1: $\left[\begin{array}{l} a := x; \ /\!\!/ 0 \\ x := a + 1; \end{array}\right.$ ‖ **T2:** $\left[\begin{array}{l} b := x; \ /\!\!/ 0 \\ x := b + 1; \end{array}\right.$ (LU) Lost Update SI: ✗	**T1:** $\left[\begin{array}{l} a := x; \ /\!\!/ 0 \\ y := 1; \end{array}\right.$ ‖ **T2:** $\left[\begin{array}{l} b := y; \ /\!\!/ 0 \\ x := 1; \end{array}\right.$ (WS) Write Skew SI: ✓	**T1:** $[y := 1;$ **T3:** $[a := x; \ /\!\!/ 0$ ‖ **T2:** $\left[\begin{array}{l} b := y; \ /\!\!/ 0 \\ x := 1; \end{array}\right.$ (WS2) Write Skew Variant SI: ✓
T1: $\left[\begin{array}{l} x := 1; \\ y := 1; \end{array}\right.$ **T3:** $[a := y; \ /\!\!/ 2$ ‖ **T2:** $\left[\begin{array}{l} b := x; \ /\!\!/ 0 \\ y := 2; \end{array}\right.$ (LU2) Lost Update Variant SI: ✗	$x := 1;$ **T1:** $[a := z;$ $b := y; \ /\!\!/ 0$ ‖ $y := 1;$ **T2:** $[c := z;$ $d := x; \ /\!\!/ 0$ (SBT) Store Buffering RSI: ✓	$x := 1;$ $y := 1;$ ‖ **T2:** $\left[\begin{array}{l} a := y; \ /\!\!/ 1 \\ b := x; \ /\!\!/ 0 \end{array}\right.$ (MPT) Message Passing RSI: ✗

Fig. 1. Litmus tests illustrating transaction anomalies and their admissibility under SI and RSI. In all tests, initially, $x = y = z = 0$. The $/\!\!/ v$ annotation next to a read records the value read.

transactions of C++. As we describe later in Sect. 2.3, it is straightforward to lift the latter restriction (2) for our lazy implementations.

For non-transactional accesses, we naturally have to pick some consistency model. For simplicity and uniformity, we pick the release/acquire (RA) subset of the C++ memory model [6,23], a well-behaved platform-independent memory model, whose compilation to x86 requires no memory fences.

Snapshot Isolation (SI). The initial model of SI in [7] is described informally in terms of a multi-version concurrent algorithm as follows. A transaction T proceeds by taking a *snapshot* S of the shared objects. The execution of T is then carried out locally: read operations query S and write operations update S. Once T completes its execution, it attempts to *commit* its changes and succeeds *only if* it is not *write-conflicted.* Transaction T is write-conflicted if another *committed* transaction T′ has written to a location also written to by T, since T recorded its snapshot. If T fails the conflict check it aborts and may restart; otherwise, it commits its changes, and its changes become visible to all other transactions that take a snapshot thereafter.

To realise this, the shared state is represented as a series of *multi-versioned* objects: each object is associated with a history of several versions at different *timestamps.* In order to obtain a snapshot, a transaction T chooses a *start-timestamp* t_0, and reads data from the committed state as of t_0, ignoring updates after t_0. That is, updates committed after t_0 are invisible to T. In order to commit, T chooses a *commit-timestamp* t_c larger than any existing start- or commit-timestamp. Transaction T is deemed write-conflicted if another transaction T′ has written to a location also written to by T *and* the commit-timestamp of T′ is in the execution interval of T ($[t_0, t_c]$).

2.1 Towards an SI Reference Implementation Without Timestamps

While the SI description above is suitable for understanding SI, it is not useful for integrating the SI model in a language such as C/C++ or Java. From a programmer's perspective, in such languages the various threads directly access the *uninstrumented* (single-versioned) shared memory; they do not access their own instrumented snapshot at a particular timestamp, which is loosely related to the snapshots of other threads. Ideally, what we would therefore like is an equivalent description of SI in terms of accesses to uninstrumented shared memory and a synchronisation mechanism such as locks.

In what follows, we present our first lock-based reference implementation for SI that does not rely on timestamps. To do this, we assume that the locations accessed by a transaction can be statically determined. Specifically, we assume that each transaction `T` is supplied with its *read set*, `RS`, and *write set*, `WS`, containing those locations read and written by `T`, respectively (a static over-approximation of these sets suffices for soundness.). As such, our first reference implementation is *prescient* [19] in that it requires *a priori* knowledge of the locations accessed by the transaction. Later in Sect. 2.3 we lift this assumption and develop an SI reference implementation that is *non-prescient* and similarly does not rely on timestamps.

Conceptually, a candidate implementation of transaction `T` would (1) obtain a snapshot of the locations read by `T`; (2) lock those locations written by `T`; (3) execute `T` *locally*; and (4) unlock the locations written. The snapshot is obtained via `snapshot(RS)` in Fig. 3 where the values of locations in `RS` are recorded in a local array `s`. The local execution of `T` is carried out by executing $(\!|\texttt{T}|\!)$ in Fig. 3, which is obtained from `T` by (i) modifying read operations to read locally from the snapshot in `s`, and (ii) updating the snapshot after each write operation. Note that the snapshot must be obtained *atomically* to reflect the memory state at a particular instance (*cf.* start-timestamp). An obvious way to ensure the snapshot atomicity is to lock the locations in the read set, obtain a snapshot, and unlock the read set. However, as we must allow for two transactions *reading* from the same location to execute in parallel, we opt for *multiple-readers-single-writer* (MRSW) locks.

Let us now try to make this general pattern more precise. As a first attempt, consider the implementation in Fig. 2a written in a simple while language, which releases all the reader locks at the end of the snapshot phase before acquiring any writer locks. This implementation is unsound as it admits the lost update (LU) anomaly in Fig. 1 disallowed under SI [11]. To understand this, consider a scheduling where T2 runs between lines 3 and 4 of T1 in Fig. 2a, which would result in T1 having read a stale value. The problem is that the writer locks on `WS` are acquired too late, allowing two conflicting transactions to run concurrently. To address this, writer locks must be acquired early enough to pre-empt the concurrent execution of write-write-conflicting transactions. Note that locks have to be acquired early even for locations only written by a transaction to avoid exhibiting a variant of the lost update anomaly (LU2).

1. `for (x ∈ RS) lock_r x` 2. `snapshot(RS);` 3. `for (x ∈ RS) unlock_r x` 4. `for (x ∈ WS) lock_w x` 5. ⦇T⦈; 6. `for (x ∈ WS) unlock_w x` (a)	1. `for (x ∈ WS) lock_w x;` 2. `for (x ∈ RS\WS) lock_r x` 3. `snapshot(RS);` 4. `for (x ∈ RS\WS) unlock_r x` 5. ⦇T⦈; 6. `for (x ∈ WS) unlock_w x` (b)	1. `for (x ∈ RS ∪ WS) lock_r x` 2. `snapshot(RS);` 3. `for (x ∈ RS ∪ WS) {` 4. `  if (x ∈ WS) promote x` 5. `  else unlock_r x; }` 6. ⦇T⦈; 7. `for (x ∈ WS) unlock_w x` (c)
Sound: ✗ allows (LU), (LU2)	Sound: ✓ Complete: ✗ disallows (WS)	Sound: ✓ Complete: ✗ disallows (WS2)

Fig. 2. Candidate SI implementations of transaction T given read/write sets RS, WS

As such, our second candidate implementation in Fig. 2b brings forward the acquisition of writer locks. Whilst this implementation is sound (and disallows lost update), it nevertheless disallows behaviours deemed valid under SI such as the write skew anomaly (WS) in Fig. 1, and is thus incomplete. The problem is that such early acquisition of writer locks not only pre-empts concurrent execution of *write-write-conflicting* transactions, but also those of *read-write-conflicting* transactions (e.g. WS) due to the exclusivity of writer locks.

To remedy this, in our third candidate implementation in Fig. 2c we first acquire weaker reader locks on all locations in RS or WS, and later *promote* the reader locks on WS to exclusive writer ones, while releasing the reader locks on RS. The promotion of a reader lock signals its intent for exclusive ownership and awaits the release of the lock by other readers before claiming it exclusively as a writer. To avoid deadlocks, we further assume that RS ∪ WS is ordered so that locks are promoted in the same order by all threads.

Although this implementation is "more complete" than the previous one, it is still incomplete as it disallows certain behaviour admitted by SI. In particular, consider a variant of the write skew anomaly (WS2) depicted in Fig. 1, which is admitted under SI, but not admitted by this implementation.

To understand why this is admitted by SI, recall the operational SI model using timestamps. Let the domain of timestamps be that of natural numbers $\mathbb{N}$. The behaviour of (WS2) can be achieved by assigning the following execution intervals for T1: $[t_0^{\mathtt{T}_1} = 2, t_c^{\mathtt{T}_1} = 2]$; T2: $[t_0^{\mathtt{T}_2} = 1, t_c^{\mathtt{T}_2} = 4]$; and T3: $[t_0^{\mathtt{T}_3} = 3, t_c^{\mathtt{T}_3} = 3]$. To see why the implementation in Fig. 2c does not admit the behaviour in (WS2), let us assume without loss of generality that x is ordered before y. Upon executing lines 3–5, (a) T1 promotes y; (b) T2 promotes x and then (c) releases the reader lock on y; and (d) T3 releases the reader lock on x. To admit the behaviour in (WS2), the release of y in (c) must occur before the promotion of y in (a) since otherwise T2 cannot read 0 for y. Similarly, the release of x in (d) must occur before its promotion in (b). On the other hand, since T3 is executed by the same thread after T1, we know that (a) occurs before (d). This however leads to circular execution: (b)→(c)→(a)→(d)→(b), which cannot be realised.

```
 0. LS := ∅;
 1. for (x ∈ RS ∪ WS) lock_r x
 2. snapshot(RS);
 3. for (x ∈ RS\WS) unlock_r x
 4. for (x ∈ WS) {
 5.   if (can-promote x) LS.add(x)
 6.   else {
 7.     for (x ∈ LS) unlock_w x
 8.     for (x ∈ WS \ LS) unlock_r x
 9.     goto line 0 }
10. }
11. ⦇T⦈;
12. for (x ∈ WS) unlock_w x
```

```
snapshot(RS) ≜ for (x ∈ RS) s_x := x

snapshot_RSI(RS) ≜
    start: for (x ∈ RS) s_x := x
           for (x ∈ RS) {
             if (s_x != x) goto start
           }
```

$$⦇\texttt{a:=x}⦈ \triangleq \texttt{a:=s}_\texttt{x}$$

$$⦇\texttt{x:=a}⦈ \triangleq \texttt{x:=a; s}_\texttt{x}\texttt{:=a}$$

$$⦇S_1 ; S_2⦈ \triangleq ⦇S_1⦈ ; ⦇S_2⦈$$

$$⦇\texttt{while(e)}\ S⦈ \triangleq \texttt{while(e)}\ ⦇S⦈$$

... and so on ...

Fig. 3. SI implementation of transaction `T` given `RS`, `WS`; the code in blue ensures deadlock avoidance. The RSI implementation (Sect. 5) is obtained by replacing `snapshot` on line 2 with $\texttt{snapshot}_{\text{RSI}}$.

To overcome this, in our final candidate execution in Fig. 3 (ignoring the code in blue), after obtaining a snapshot, we *first* release the reader locks on `RS`, and *then* promote the reader locks on `WS`, rather than simultaneously in one pass. As we demonstrate in Sect. 4, the implementation in Fig. 3 is both *sound and complete* against its declarative SI specification.

Avoiding Deadlocks. As two distinct reader locks on `x` may simultaneously attempt to promote their locks, promotion is done on a 'first-come-first-served' basis to avoid *deadlocks*. A call to `can-promote x` by reader r thus returns a boolean denoting either (i) successful promotion (true); or (ii) failed promotion as another reader r' is currently promoting a lock on `x` (false). In the latter case, r must release its reader lock on `x` to ensure the successful promotion of `xl` by r' and thus avoid deadlocks. To this end, our implementation in Fig. 3 includes a deadlock avoidance mechanism (code in blue) as follows. We record a list `LS` of those locks on the write set that have been successfully promoted so far. When promoting a lock on `x` succeeds (line 5), the `LS` is extended with `x`. On the other hand, when promoting `x` fails (line 6), all those locks promoted so far (i.e. in `LS`) as well as those yet to be promoted (i.e. in `WS` \`LS`) are released and the transaction is restarted.

Remark 1. Note that the deadlock avoidance code in blue does not influence the correctness of the implementation in Fig. 3, and is merely included to make the reference implementation more realistic. In particular, the implementation without the deadlock avoidance code is both sound and complete against the SI specification, provided that the conditional `can-promote` call on line 5 is replaced by the blocking `promote` call.

Avoiding Over-Synchronisation Due to MRSW Locks. Consider the store buffering program (SBT) shown in Fig. 1. If, for a moment, we ignore transactional accesses, our underlying memory model (RA)—as well as all other weak memory models—allows the annotated weak behaviour. Intuitively, placing the two transactions that only *read* z in (SBT) should still allow the weak behaviour since the two transactions do not need to synchronise in any way. Nevertheless, most MRSW lock implementations forbid this outcome because they use a single global counter to track the number of readers that have acquired the lock, which inadvertently also synchronises the readers with one another. As a result, the two read-only transactions act as memory fences forbidding the weak outcome of (SBT). To avoid such synchronisation, in the technical appendix [31] we provide a different MRSW implementation using a separate location for each thread so that reader lock acquisitions do not synchronise.

To keep the presentation simple, we henceforth assume an abstract specification of a MRSW lock library providing operations for acquiring/releasing reader/writer locks, as well as promoting reader locks to writer ones. We require that (1) calls to writer locks (to acquire, release or promote) *synchronise* with all other calls to the lock library; and (2) writer locks provide *mutual exclusion* while held. We formalise these notions in Sect. 4. These requirements do not restrict synchronisation between *two read* lock calls: two read lock calls may or may not synchronise. Synchronisation between read lock calls is relevant only for the completeness of our RSI implementation (handling mixed-mode code); for that result, we further require that (3) read lock calls not synchronise.

2.2 Handling Racy Mixed-Mode Accesses

Let us consider what happens when data accessed by a transaction is modified concurrently by an uninstrumented atomic non-transactional write. Since such writes do not acquire any locks, the snapshots taken may include values written by non-transactional accesses. The result of the snapshot then depends on the order in which the variables are read. Consider the (MPT) example in Fig. 1. In our implementation, if in the snapshot phase y is read before x, then the annotated weak behaviour is not possible because the underlying model (RA) disallows this weak "message passing" behaviour. If, however, x is read before y, then the weak behaviour is possible. In essence, this means that the SI implementation described so far is of little use when there are races between transactional and non-transactional code. Technically, our SI implementation violates *monotonicity* with respect to wrapping code inside a transaction. The weak behaviour of the (MPT) example is disallowed by RA if we remove the transaction block T2, and yet it is exhibited by our SI implementation with the transaction block.

To get monotonicity under RA, it suffices for the snapshots to read the variables in the same order they are accessed by the transactions. Since a static calculation of this order is not always possible, following [30], we achieve this by reading each variable twice. In more detail, our $\texttt{snapshot}_{\text{RSI}}$ implementation in Fig. 3 takes *two* snapshots of the locations read by the transaction, and checks that they both 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. As we demonstrate in Sect. 5, our RSI implementation is both *sound and complete* against our proposed declarative semantics for RSI. There is however one caveat: since equality of values is used to determine whether the two snapshots agree, we will miss cases where different non-transactional writes to a location write the same value. In our formal development (see Sect. 5), we thus assume that if multiple non-transactional writes write the same value to the same location, they cannot race with the same transaction. Note that this assumption cannot be lifted without instrumenting non-transactional writes, and thus impeding performance substantially. That is, to lift this restriction we must instead replace every non-transactional *write* `x:=v` with `lock_w x; x:=v; unlock_w x`.

2.3 Non-prescient Reference Implementations Without Timestamps

Recall that the SI and RSI implementations in Sect. 2.1 are prescient in that they require knowledge of the read and write sets of transactions beforehand. In what follows we present alternative SI and RSI implementations that are *non-prescient*.

Non-prescient SI Reference Implementation. In Fig. 4 we present a *lazy* lock-based reference implementation for SI. This implementation is *non-prescient* and does not require *a priori* knowledge of the read set `RS` and the write set `WS`. Rather, the `RS` and `WS` are computed on the fly as the execution of the transaction unfolds. As with the SI implementation in Fig. 3, this implementation does not rely on timestamps and uses MRSW locks to synchronise concurrent accesses to shared data. As before, the implementation consults a local *snapshot* at `s` for read operations. However, unlike the eager implementation in Fig. 3 where transactional writes are performed *in-place*, the implementation in Fig. 4 is *lazy* in that it logs the writes in the local array `s` and propagates them to memory at commit time, as we describe shortly.

Ignoring the code in blue, the implementation in Fig. 4 proceeds with initialising `RS` and `WS` with $\emptyset$ (line 1); it then populates the local snapshot array at `s` with initial value $\perp$ for each location `x` (line 2). It then executes $(\!|\texttt{T}|\!)$ which is obtained from `T` as follows. For each *read* operation `a:=x` in `T`, first the value of `s[x]` is inspected to ensure it contains a snapshot of `x`. If this is not the case (i.e. $\texttt{x} \notin \texttt{RS} \cup \texttt{WS}$), a reader lock on `x` is acquired, a snapshot of `x` is recorded in `s[x]`, and the read set `RS` is extended with `x`. The snapshot value in `s[x]` is subsequently returned in `a`. Analogously, for each write operation `x:=a`, the `WS` is extended with `x`, and the written value is lazily logged in `s[x]`. Recall from our candidate executions in Fig. 2 that to ensure implementation correctness, for each written location `x`, the implementation must first acquire a reader lock on `x`, and subsequently promote it to a writer lock. As such, for each write operation in `T`, the implementation first checks if a reader lock for `x` has been acquired (i.e. $\texttt{x} \in \texttt{RS} \cup \texttt{WS}$) and obtains one if this is not the case.

```
 0.  LS:=∅;
 1.  RS:=∅; WS:=∅;
 2.  for (x∈Locs) s[x]:=⊥
 3.  ⦇T⦈;
 4.  for (x∈RS\WS) unlock_r x
 5.  for (x∈WS) {
 6.    if (can-promote x) LS.add(x)
 7.    else {
 8.      for (x∈LS) unlock_w x
 9.      for (x∈WS \ LS) unlock_r x
10.      goto line 0 }  }
11.  for (x∈WS) x:=s[x]
12.  for (x∈WS) unlock_w x
```

```
    ⦇a:=x⦈ ≜ if (x∉RS∪WS) {
                 lock_r x; RS.add(x);
                 s[x]:=x;
             }
             a:=s[x];

    ⦇x:=a⦈ ≜ if (x∉RS∪WS) lock_r x;
             WS.add(x); s[x]:=a;

  ⦇S1;S2⦈ ≜ ⦇S1⦈;⦇S2⦈

⦇while(e) S⦈ ≜ while(e) ⦇S⦈

    ... and so on ...
```

Fig. 4. Non-prescient SI implementation of transaction T with RS and WS computed on the fly; the code in blue ensures deadlock avoidance.

Once the execution of ⦇T⦈ is completed, the implementation proceeds to *commit* the transaction. To this end, the reader locks on RS are released (line 4), reader locks on WS are promoted to writer ones (line 6), the writes logged in s are propagated to memory (line 11), and finally the writer locks on WS are released (line 12). As we demonstrate later in Sect. 4, the implementation in Fig. 4 is both sound and complete against the declarative SI specification.

Note that the implementation in Fig. 4 is optimistic in that it logs the writes performed by the transaction in the local array s and propagates them to memory at commit time, rather than performing the writes *in-place* as with its pessimistic counterpart in Fig. 3. As before, the code in blue ensures deadlock avoidance and is identical to its counterpart in Fig. 3. As before, this deadlock avoidance code does not influence the correctness of the implementation and is merely included to make the reference implementation more practical.

Non-prescient RSI Reference Implementation. In Fig. 5 we present a *lazy* lock-based reference implementation for RSI. As with its SI counterpart, this implementation is non-prescient and computes the RS and WS on the fly. As before, the implementation does not rely on timestamps and uses MRSW locks to synchronise concurrent accesses to shared data. Similarly, the implementation consults the local *snapshot* at s for read operations, whilst logging write operations lazily in a *write sequence* at wseq, as we describe shortly.

Recall from the RSI implementation in Sect. 2.1 that to ensure snapshot validity, each location is read twice to preclude intermediate non-transactional writes. As such, when writing to a location x, the initial value read (recorded in s) must not be *overwritten* by the transaction to allow for subsequent validation of the snapshot. To this end, for each location x, the snapshot array s contains a *pair* of values, (r, c), where r denotes the snapshot value (initial value read), and c denotes the current value which may have overwritten the snapshot value.

Recall that under weak isolation, the intermediate values written by a transaction may be observed by non-transactional reads. For instance, given the

```
 0.  LS:=∅;
 1.  RS:=∅; WS:=∅; wseq:=[];
 2.  for (x∈Locs) s[x]:=(⊥,⊥)
 3.  ⦇T⦈;
 4.  for (x∈RS) {(r,-):=s[x];
 5.    if (x!=r) { //read x again
 6.      for (x∈RS∪WS) unlock_r x
 7.      goto line 0 } }
 8.  for (x∈RS\WS) unlock_r x
 9.  for (x∈WS) {
10.    if (can-promote x) LS.add(x)
11.    else {
12.      for (x∈LS) unlock_w x
13.      for (x∈WS\LS) unlock_r x
14.      goto line 0 }  }
15.  for ((x,v)∈wseq) x:=v
16.  for (x∈WS) unlock_w x
```

```
    ⦇a:=x⦈ ≜ if (x∉RS∪WS) {
                lock_r x; RS.add(x);
                r:=x; s[x]:=(r,r);
             } (-,c):=s[x]; a:=c;

    ⦇x:=a⦈ ≜ if (x∉RS∪WS) lock_r x
             WS.add(x);
             (r,-):=s[x]; s[x]:=(r,a);
             wseq:=wseq++[(x,a)];

  ⦇S1;S2⦈ ≜ ⦇S1⦈;⦇S2⦈

⦇while(e) S⦈ ≜ while(e) ⦇S⦈

    ... and so on ...
```

Fig. 5. Non-prescient RSI implementation of transaction `T` with `RS` and `WS` computed on the fly; the code in blue ensures deadlock avoidance.

T: $\left[x := 1; x := 2 \;\middle|\middle|\; a := x\right.$ program, the non-transactional read $a := x$, may read either 1 or 2 for x. As such, at commit time, it is not sufficient solely to propagate the last written value (in program order) to each location (e.g. to propagate only the $x := 2$ write in the example above). Rather, to ensure implementation completeness, one must propagate all written values to memory, in the order they appear in the transaction body. To this end, we track the values written by the transaction as a (FIFO) *write sequence* at location `wseq`, containing items of the form (x, v), denoting the location written (x) and the associated value (v).

Ignoring the code in blue, the implementation in Fig. 5 initialises `RS` and `WS` with $\emptyset$, initialises `wseq` as an empty sequence `[]` (line 1), and populates the local snapshot array `s` with initial value $(\perp, \perp)$ for each location `x` (line 2). It then executes ⦇`T`⦈, obtained from `T` in an analogous manner to that in Fig. 4. For every read `a:=x` in ⦇`T`⦈, the current value recorded for `x` in `s` (namely `c` when `s[x]` holds `(-,c)`) is returned in `a`. Dually, for every write `x:=a` in ⦇`T`⦈, the current value recorded for `x` in `s` is updated to `a`, and the write is logged in the write sequence `wseq` by appending `(x,a)` to it.

Upon completion of ⦇`T`⦈, the snapshot in `s` is *validated* (lines 4–7). Each location `x` in `RS` is thus read again and its value is compared against the snapshot value in `s[x]`. If validation fails (line 5), the locks acquired are released (line 6) and the transaction is restarted (line 7).

If validation succeeds, the transaction is committed: the reader locks on `RS` are released (line 8), the reader locks on `WS` are promoted (line 10), the writes in `wseq` are propagated to memory in FIFO order (line 15), and finally the writer locks on `WS` are released (line 16).

As we show in Sect. 5, the implementation in Fig. 5 is both sound and complete against our proposed declarative specification for RSI. As before, the code in blue ensures deadlock avoidance; it does not influence the implementation correctness and is merely included to make the implementation more practical.

Supporting Explicit Abort Instructions. It is straightforward to extend the lazy implementations in Figs. 4 and 5 to handle transactions containing explicit abort instructions. More concretely, as the effects (writes) of a transaction are logged locally and are not propagated to memory until commit time, upon reaching an `abort` in $(\!|T|\!)$ no roll-back is necessary, and one can simply release the locks acquired so far and return. That is, one can extend $(\!|.|\!)$ in Figs. 4 and 5, and define $(\!|\texttt{abort}|\!) \triangleq$ `for (x` $\in$ `RS` $\cup$ `WS) unlock_r x; return`.

3 A Declarative Framework for STM

We present the notational conventions used in the remainder of this article, and describe a general framework for declarative concurrency models. Later in this article, we present SI, its extension with non-transactional accesses, and their lock-based implementations as instances of this general definition.

Notation. Given a relation r on a set A, we write $\mathsf{r}^?$, r^+ and r^* for the reflexive, transitive and reflexive-transitive closure of r, respectively. We write r^{-1} for the inverse of r; $\mathsf{r}|_A$ for $\mathsf{r} \cap (A \times A)$; $[A]$ for the identity relation on A, i.e. $\{(a,a) \mid a \in A\}$; $\mathsf{irreflexive}(r)$ for $\nexists a.\ (a,a) \in r$; and $\mathsf{acyclic}(\mathsf{r})$ for $\mathsf{irreflexive}(\mathsf{r}^+)$. Given two relations r_1 and 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 r is a strict partial order, we write $\mathsf{r}|_{\text{imm}}$ for the *immediate* edges in r: $\{(a,b) \in \mathsf{r} \mid \nexists c.\ (a,c) \in \mathsf{r} \wedge (c,b) \in \mathsf{r}\}$.

Assume finite sets of *locations* $\textsc{Loc}$; *values* $\textsc{Val}$; *thread identifiers* $\textsc{TId}$, and *transaction identifiers* $\textsc{TXId}$. We use x, y, z to range over locations, v over values, τ over thread identifiers, and ξ over transaction identifiers.

Definition 1 (Events). *An event is a tuple $\langle n, \tau, \xi, l\rangle$, where $n \in \mathbb{N}$ is an event identifier, $\tau \in \textsc{TId} \uplus \{0\}$ is a thread identifier (0 is used for initialisation events), $\xi \in \textsc{TXId} \uplus \{0\}$ is a transaction identifier (0 is used for non-transactional events), and l is an event label that takes one of the following forms:*

- *A memory access label:* $\mathtt{R}(x,v)$ *for* reads*;* $\mathtt{W}(x,v)$ *for* writes*; and* $\mathtt{U}(x,v_r,v_w)$ *for* updates.
- *A lock label:* $\mathtt{RL}(x)$ *for* reader lock acquisition*;* $\mathtt{RU}(x)$ *for* reader lock release*;* $\mathtt{WL}(x)$ *for* writer lock acquisition*;* $\mathtt{WU}(x)$ *for* writer lock release*; and* $\mathtt{PL}(x)$ *for* reader to writer lock promotion.

We typically use a, b, and e to range over events. The functions `tid`, `tx`, `lab`, `typ`, `loc`, $\mathtt{val}_\mathtt{r}$ *and* $\mathtt{val}_\mathtt{w}$ *respectively project the thread identifier, transaction identifier, label, type (in $\{\mathtt{R},\mathtt{W},\mathtt{U},\mathtt{RL},\mathtt{RU},\mathtt{WL},\mathtt{WU},\mathtt{PL}\}$), location, and read/written values of an event, where applicable. We assume only reads and writes are used in transactions ($\mathtt{tx}(a) \neq 0 \implies \mathtt{typ}(a) \in \{\mathtt{R},\mathtt{W}\}$).*

Given a relation r on events, we write r_{loc} for $\{(a,b) \in \mathsf{r} \mid \mathtt{loc}(a) = \mathtt{loc}(b)\}$. Analogously, given a set A of events, we write A_x for $\{a \in A \mid \mathtt{loc}(a) = x\}$.

Definition 2 (Execution graphs). *An* execution graph*, G, is a tuple of the form* $(E, \mathsf{po}, \mathsf{rf}, \mathsf{mo}, \mathsf{lo})$*, where:*

- *E is a set of events, assumed to contain a set E_0 of initialisation events, consisting of a write event with label $\mathtt{W}(x, 0)$ for every $x \in \text{Loc}$. The sets of* read events *in E is denoted by $\mathcal{R} \triangleq \{e \in E \mid \mathtt{typ}(e) \in \{\mathtt{R}, \mathtt{U}\}\}$;* write events *by $\mathcal{W} \triangleq \{e \in E \mid \mathtt{typ}(e) \in \{\mathtt{W}, \mathtt{U}\}\}$;* update events *by $\mathcal{U} \triangleq \mathcal{R} \cap \mathcal{W}$; and* lock events *by $\mathcal{L} \triangleq \{e \in E \mid \mathtt{typ}(e) \in \{\mathtt{RL}, \mathtt{RU}, \mathtt{WL}, \mathtt{WU}, \mathtt{PL}\}\}$. The sets of reader lock acquisition and release events, $\mathcal{RL}$ and $\mathcal{RU}$, writer lock acquisition and release events, $\mathcal{WL}$ and $\mathcal{WU}$, and lock promotion events $\mathcal{PL}$ are defined analogously. The set of transactional events in E is denoted by $\mathcal{T}$ ($\mathcal{T} \triangleq \{e \in E \mid \mathtt{tx}(e) \neq 0\}$); and the set of non-transactional events is denoted by $\mathcal{NT}$ ($\mathcal{NT} \triangleq E \setminus \mathcal{T}$).*
- *$\mathsf{po} \subseteq E \times E$ denotes the* 'program-order' relation*, defined as a disjoint union of strict total orders, each ordering the events of one thread, together with $E_0 \times (E \setminus E_0)$ that places the initialisation events before any other event. We assume that events belonging to the same transaction are ordered by po, and that any other event po-between them also belongs to the same transaction.*
- *$\mathsf{rf} \subseteq \mathcal{W} \times \mathcal{R}$ denotes the* 'reads-from' relation*, defined between write and read events of the same location with matching read and written values; it is total and functional on reads, i.e. every read is related to exactly one write.*
- *$\mathsf{mo} \subseteq \mathcal{W} \times \mathcal{W}$ denotes the* 'modification-order' relation*, defined as a disjoint union of strict total orders, each ordering the write events on one location.*
- *$\mathsf{lo} \subseteq \mathcal{L} \times \mathcal{L}$ denotes the* 'lock-order' relation*, defined as a disjoint union of strict orders, each of which (partially) ordering the lock events to one location.*

In the context of an execution graph $G = (E, \mathsf{po}, \mathsf{rf}, \mathsf{mo}, \mathsf{lo})$—we often use "$G$." as a prefix to make this explicit—the *'same-transaction' relation*, $\mathsf{st} \in \mathcal{T} \times \mathcal{T}$, is the equivalence relation given by $\mathsf{st} \triangleq \{(a, b) \in \mathcal{T} \times \mathcal{T} \mid \mathtt{tx}(a) = \mathtt{tx}(b)\}$. Given a relation $\mathsf{r} \subseteq E \times E$, we write r_T for lifting r to transaction classes: $\mathsf{r}_\mathsf{T} \triangleq \mathsf{st}; (\mathsf{r} \setminus \mathsf{st}); \mathsf{st}$. For instance, when $(w, r) \in \mathsf{rf}$, w is a transaction ξ_1 event and r is a transaction ξ_2 event, then all events in ξ_1 are rf_T-related to all events in ξ_2. Analogously, we write r_I to restrict r to its *intra-transactional* edges (within a transaction): $\mathsf{r}_\mathsf{I} \triangleq \mathsf{r} \cap \mathsf{st}$; and write r_E to restrict r to its *extra-transactional* edges (outside a transaction): $\mathsf{r}_\mathsf{E} \triangleq \mathsf{r} \setminus \mathsf{st}$. Lastly, the 'reads-before' relation is defined by $\mathsf{rb} \triangleq (\mathsf{rf}^{-1}; \mathsf{mo}) \setminus [E]$. Intuitively, rb relates a read r to all writes w that are mo-after the write r reads from; i.e. when $(w', r) \in \mathsf{rf}$ and $(w', w) \in \mathsf{mo}$, then $(r, w) \in \mathsf{rb}$. In the transactional literature, this is known as the *anti-dependency* relation [3,4].

Execution graphs of a given program represent traces of shared memory accesses generated by the program. The set of execution graphs associated with programs written in our while language can be straightforwardly defined by induction over the structure of programs as in e.g. [35]. Each execution of a program P has a particular program *outcome*, prescribing the final values of local variables in each thread. In this initial stage, the execution outcomes are almost unrestricted as there are very few constraints on the rf, mo and lo relations. Such restrictions and thus the permitted outcomes of a program are determined by defining the set of *consistent* executions, which is defined separately for each model we consider. Given a program P and a model M, the set $\mathsf{outcomes}_M(P)$ collects the outcomes of every M-consistent execution of P.

4 Snapshot Isolation (SI)

We present a declarative specification of SI and demonstrate that the SI implementations presented in Figs. 3 and 4 are both sound and complete with respect to the SI specification.

In [11] Cerone and Gotsman developed a declarative specification for SI using dependency graphs [3,4]. Below we adapt their specification to the notation of Sect. 3. As with [11], throughout this section, we take *SI execution graphs* to be those in which $E = \mathcal{T} \subseteq (\mathcal{R} \cup \mathcal{W}) \setminus \mathcal{U}$. That is, the SI model handles transactional code only, consisting solely of read and write events (excluding updates).

Definition 3 (SI consistency [11]). *An SI execution* $G = (E, \mathsf{po}, \mathsf{rf}, \mathsf{mo}, \mathsf{lo})$ *is* SI-consistent *if the following conditions hold:*

– $\mathsf{rf}_{\mathrm{I}} \cup \mathsf{mo}_{\mathrm{I}} \cup \mathsf{rb}_{\mathrm{I}} \subseteq \mathsf{po}$ (INT)
– $\mathsf{acyclic}((\mathsf{po}_{\mathsf{T}} \cup \mathsf{rf}_{\mathsf{T}} \cup \mathsf{mo}_{\mathsf{T}}); \mathsf{rb}_{\mathsf{T}}{}^{?})$ (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 equivalent formulation of SI-consistency which will serve as the basis of our extension with non-transactional accesses in Sect. 5.

Proposition 1. *An SI execution* $G = (E, \mathsf{po}, \mathsf{rf}, \mathsf{mo}, \mathsf{lo})$ *is* SI-consistent *if and only if* INT *holds and the 'SI-happens-before' relation* $\mathsf{si\text{-}hb} \triangleq (\mathsf{po}_{\mathsf{T}} \cup \mathsf{rf}_{\mathsf{T}} \cup \mathsf{mo}_{\mathsf{T}} \cup \mathsf{si\text{-}rb})^{+}$ *is irreflexive, where* $\mathsf{si\text{-}rb} \triangleq [\mathcal{R}_{\mathsf{E}}]; \mathsf{rb}_{\mathsf{T}}; [\mathcal{W}]$ *and* $\mathcal{R}_{\mathsf{E}} \triangleq \{r \mid \exists w.\ (w, r) \in \mathsf{rf}_{\mathsf{E}}\}$.

Proof. The full proof is given in the technical appendix [31].

Intuitively, SI-happens-before orders events of different transactions; this order is due to either the program order (po_{T}), or synchronisation enforced by the implementation ($\mathsf{rf}_{\mathsf{T}} \cup \mathsf{mo}_{\mathsf{T}} \cup \mathsf{si\text{-}rb}$). By contrast, events of the same transaction are unordered, as the implementation may well execute them in a different order (in particular, by taking a snapshot, it executes external reads before the writes).

In more detail, the rf_T corresponds to transactional synchronisation due to *causality*, i.e. when one transaction $\mathtt{T}_2$ observes an effect of an earlier transaction $\mathtt{T}_1$. The inclusion of rf_T ensures that $\mathtt{T}_2$ cannot read from $\mathtt{T}_1$ without observing its *entire* effect. This in turn ensures that transactions exhibit 'all-or-nothing' behaviour: they 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 x from $\mathtt{T}_1$ but read y from an earlier (in 'happens-before' order) transaction $\mathtt{T}_0$.

The mo_T corresponds to transactional synchronisation due to *write-write conflicts*. Its inclusion enforces write-conflict-freedom of SI transactions: if $\mathtt{T}_1$ and $\mathtt{T}_2$ both write to x via events w_1 and w_2 such that $(w_1, w_2) \in \mathsf{mo}$, then $\mathtt{T}_1$ must commit before $\mathtt{T}_2$, and thus its entire effect must be visible to $\mathtt{T}_2$.

To understand $\mathsf{si\text{-}rb}$, first note that $\mathcal{R}_\mathsf{E}$ denotes the *external* transactional reads (i.e. those reading a value written by another transaction). That is, the $\mathcal{R}_\mathsf{E}$ are the read events that get their values from the transactional snapshot phases. By contrast, internal reads (those reading a value written by the same transaction) happen only after the snapshot is taken. Now let there be an rb_T edge between two transactions, $\mathtt{T}_1$ and $\mathtt{T}_2$. This means there exist a read event r of $\mathtt{T}_1$ and a write event w of $\mathtt{T}_2$ such that $(r, w) \in \mathsf{rb}$; i.e. there exists w' such that $(w', r) \in \mathsf{rf}$ and $(w', w) \in \mathsf{mo}$. If r reads internally (i.e. w' is an event in $\mathtt{T}_1$), then $\mathtt{T}_1$ and $\mathtt{T}_2$ are conflicting transactions and as accounted by mo_T described above, all events of $\mathtt{T}_1$ happen before those of $\mathtt{T}_2$. Now, let us consider the case when r reads externally (w' is not in $\mathtt{T}_1$). From the timestamped model of SI, there exists a start-timestamp $t_0^{\mathtt{T}_1}$ as of which the $\mathtt{T}_1$ snapshot (all its external reads including r) is recorded. Similarly, there exists a commit-timestamp $t_c^{\mathtt{T}_2}$ as of which the updates of $\mathtt{T}_2$ (including w) are committed. Moreover, since $(r, w) \in \mathsf{rb}$ we know $t_0^{\mathtt{T}_1} < t_c^{\mathtt{T}_2}$ (otherwise r must read the value written by w and not w'). That is, we know all events in the snapshot of $\mathtt{T}_1$ (i.e. all external reads in $\mathtt{T}_1$) happen before all writes of $\mathtt{T}_2$.[2]

We use the declarative framework in Sect. 3 to formalise the semantics of our implementation. Here, our programs include only non-transactional code, and thus *implementation execution graphs* are taken as those in which $\mathcal{T} = \emptyset$. Furthermore, we assume that locks in implementation programs are used in a *well-formed* manner: the sequence of lock events for *each location*, in each thread (following po), should match (a prefix of) the regular expression $(\mathtt{RL}\cdot\mathtt{RU} \mid \mathtt{WL}\cdot\mathtt{WU} \mid \mathtt{RL}\cdot\mathtt{PL}\cdot\mathtt{WU})^*$. For instance, a thread never releases a lock, without having acquired it earlier in the program. As a consistency predicate on execution graphs, we use the C11 release/acquire consistency augmented with certain constraints on lock events.

Definition 4. *An implementation execution graph* $G = (E, \mathsf{po}, \mathsf{rf}, \mathsf{mo}, \mathsf{lo})$ *is* RA-consistent *if the following hold, where* $\mathsf{hb} \triangleq (\mathsf{po} \cup \mathsf{rf} \cup \mathsf{lo})^+$ *denotes the 'RA-happens-before' relation:*

- $\forall x.\, \forall a \in \mathcal{WL}_x \cup \mathcal{WU}_x \cup \mathcal{PL}_x, b \in \mathcal{L}_x.\ a = b \vee (a, b) \in \mathsf{lo} \vee (b, a) \in \mathsf{lo}$ (WSync)
- $[\mathcal{WL} \cup \mathcal{PL}]; (\mathsf{lo} \setminus \mathsf{po}); [\mathcal{L}] \subseteq \mathsf{po}; [\mathcal{WU}]; \mathsf{lo}$ (WEx)

[2] By taking rb_T instead of $\mathsf{si\text{-}rb}$ in Proposition 1 one obtains a characterisation of *serialisability*.

– $[\mathcal{RL}]; (\mathsf{lo} \setminus \mathsf{po}); [\mathcal{WL} \cup \mathcal{PL}] \subseteq \mathsf{po}; [\mathcal{RU} \cup \mathcal{PL}]; \mathsf{lo}$ (RShare)
– $\mathsf{acyclic}(\mathsf{hb}_{loc} \cup \mathsf{mo} \cup \mathsf{rb})$ (Acyc)

The (WSync) states that write lock calls (to acquire, release or promote) *synchronise* with all other calls to the same lock.

The next two constraints ensure the ‘single-writer-multiple-readers’ paradigm. In particular, (WEx) states that write locks provide *mutual exclusion* while held: any lock event l of thread τ lo-after a write lock acquisition or promotion event l' of another thread τ', is lo-after a subsequent write lock release event u of τ' (i.e. $(l', u) \in \mathsf{po}$ and $(u, l) \in \mathsf{lo}$). As such, the lock cannot be acquired (in read or write mode) by another thread until it has been released by its current owner.

The (RShare) analogously states that once a thread acquires a lock in read mode, the lock cannot be acquired in write mode by other threads until it has either been released, or promoted to a writer lock (and subsequently released) by its owner. Note that this does not preclude other threads from simultaneously acquiring the lock in read mode. In the technical appendix [31] we present two MRSW lock implementations that satisfy the conditions outlined above.

The last constraint (Acyc) is that of C11 RA consistency [23], with the hb relation extended with lo.

Remark 2. Our choice of implementing the SI STMs on top of the RA fragment is purely for presentational convenience. Indeed, it is easy to observe that execution graphs of $(\!|P|\!)$ are data race free, and thus, Acyc could be replaced by any condition that implies $\forall x.\ ([\mathcal{W}_x]; (\mathsf{po} \cup \mathsf{lo})^+; [\mathcal{W}_x]; (\mathsf{po} \cup \mathsf{lo})^+; [\mathcal{R}_x]) \cap \mathsf{rf} = \emptyset$ and that is implied by $\mathsf{acyclic}(\mathsf{po} \cup \mathsf{rf} \cup \mathsf{lo} \cup \mathsf{mo} \cup \mathsf{rb})$. In particular, the C11 non-atomic accesses or sequentially consistent accesses may be used.

We next show that our SI implementations in Figs. 3 and 4 are *sound and complete* with respect to the declarative specification given above. The proofs are non-trivial and the full proofs are given in the technical appendix [31].

Theorem 1 (Soundness and completeness). *Let P be a transactional program; let $(\!|P|\!)_{\mathrm{E}}$ denote its eager implementation as given in Fig. 3 and $(\!|P|\!)_{\mathrm{L}}$ denote its lazy implementation as given in Fig. 4. Then:*

$$\mathsf{outcomes}_{\mathrm{SI}}(P) = \mathsf{outcomes}_{\mathrm{RA}}((\!|P|\!)_{\mathrm{E}}) = \mathsf{outcomes}_{\mathrm{RA}}((\!|P|\!)_{\mathrm{L}})$$

Proof. The full proofs for both implementations is given in the technical appendix [31].

Stronger MRSW Locks. As noted in Sect. 2, for both (prescient and non-prescient) SI implementations our soundness and completeness proofs show that the same result holds for a stronger lock specification, in which reader locks synchronise as well. Formally, this specification is obtained by adding the following to Definition 4:

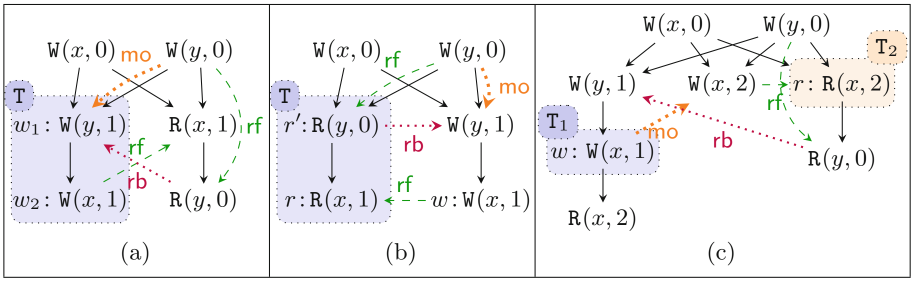

Fig. 6. RSI-inconsistent executions due to (a) rsi-po; (b) $[\mathcal{NT}]; \mathsf{rf}; \mathsf{st}$; (c) $(\mathsf{mo}; \mathsf{rf})_{\mathsf{T}}$

– $\forall x.\ \forall a, b \in \mathcal{RL}_x \cup \mathcal{RU}_x.\ a = b \vee (a, b) \in \mathsf{lo} \vee (b, a) \in \mathsf{lo}$ (RSync)

Soundness of this stronger specification ($\mathsf{outcomes}_{\mathrm{RA}}(\llparenthesis P \rrparenthesis_{\mathrm{X}}) \subseteq \mathsf{outcomes}_{\mathrm{SI}}(P)$ for $\mathrm{X} \in \{\mathrm{E}, \mathrm{L}\}$) follows immediately from Theorem 1. Completeness ($\mathsf{outcomes}_{\subseteq}(P)$ $\mathrm{SI}\mathsf{outcomes}_{\mathrm{RA}}(\llparenthesis P \rrparenthesis_{\mathrm{X}})$ for $\mathrm{X} \in \{\mathrm{E}, \mathrm{L}\}$), however, is more subtle, as we need to additionally satisfy (RSync) when constructing lo. While we can do so for SI, it is essential for the completeness of our RSI implementations that reader locks not synchronise, as shown by (SBT) in Sect. 2.

In the technical appendix [31] we present two MRSW lock implementations *sound* against the lo conditions in Definition 4. Additionally, the first implementation is *complete* against the conditions of Definition 4 augmented with (RSync), whilst the second is complete against the conditions of Definition 4 alone.

5 Robust Snapshot Isolation (RSI)

We explore the semantics of SI STMs in the presence of non-transactional code with *weak isolation* guarantees (see Sect. 2). We refer to this model as *robust snapshot isolation* (RSI), due to its ability to provide SI guarantees between transactions even in the presence of non-transactional code. We propose the first declarative specification of RSI programs and develop two lock-based reference implementations that are both *sound and complete* against our proposed specification.

A Declarative Specification of RSI STMs. We formulate a declarative specification of RSI semantics by adapting the SI semantics in Proposition 1 to account for non-transactional accesses. To specify the abstract behaviour of RSI programs, *RSI execution graphs* are taken to be those in which $\mathcal{L} = \emptyset$. Moreover, as with SI graphs, RSI execution graphs are those in which $\mathcal{T} \subseteq (\mathcal{R} \cup \mathcal{W}) \setminus \mathcal{U}$. That is, RSI transactions comprise solely read and write events, excluding updates.

Definition 5 (RSI consistency). *An execution* $G = (E, \mathsf{po}, \mathsf{rf}, \mathsf{mo}, \mathsf{lo})$ *is* RSI-consistent *iff* INT *holds and* $\mathsf{acyclic}(\mathsf{rsi\text{-}hb}_{loc} \cup \mathsf{mo} \cup \mathsf{rb})$*, where* $\mathsf{rsi\text{-}hb} \triangleq (\mathsf{rsi\text{-}po} \cup \mathsf{rsi\text{-}rf} \cup \mathsf{mo}_{\mathsf{T}} \cup \mathsf{si\text{-}rb})^{+}$ *is the 'RSI-happens-before' relation, with* $\mathsf{rsi\text{-}po} \triangleq (\mathsf{po} \setminus \mathsf{po}_{\mathrm{I}}) \cup [\mathcal{W}]; \mathsf{po}_{\mathrm{I}}; [\mathcal{W}]$ *and* $\mathsf{rsi\text{-}rf} \triangleq (\mathsf{rf}; [\mathcal{NT}]) \cup ([\mathcal{NT}]; \mathsf{rf}; \mathsf{st}) \cup \mathsf{rf}_{\mathsf{T}} \cup (\mathsf{mo}; \mathsf{rf})_{\mathsf{T}}$.

As with SI and RA, we characterise the set of executions admitted by RSI as graphs that lack cycles of certain shapes. To account for non-transactional accesses, similar to RA, we require $\mathsf{rsi\text{-}hb}_{loc} \cup \mathsf{mo} \cup \mathsf{rb}$ to be acyclic (recall that $\mathsf{rsi\text{-}hb}_{loc} \triangleq \{(a, b) \in \mathsf{rsi\text{-}hb} \mid \mathtt{loc}(a) = \mathtt{loc}(b)\}$). The RSI-happens-before relation $\mathsf{rsi\text{-}hb}$ includes both the synchronisation edges enforced by the transactional implementation (as in $\mathsf{si\text{-}hb}$), and those due to non-transactional accesses (as in hb of the RA consistency). The $\mathsf{rsi\text{-}hb}$ relation itself is rather similar to $\mathsf{si\text{-}hb}$. In particular, the mo_T and $\mathsf{si\text{-}rb}$ subparts can be justified as in $\mathsf{si\text{-}hb}$; the difference between the two lies in $\mathsf{rsi\text{-}po}$ and $\mathsf{rsi\text{-}rf}$.

To justify $\mathsf{rsi\text{-}po}$, recall from Sect. 4 that $\mathsf{si\text{-}hb}$ includes po_T. The $\mathsf{rsi\text{-}po}$ is indeed a strengthening of po_T to account for non-transactional events: it additionally includes (i) po to and from non-transactional events; and (ii) po between two *write* events in a transaction. We believe (i) comes as no surprise to the reader; for (ii), consider the execution graph in Fig. 6a, where transaction $\mathtt{T}$ is denoted by the dashed box labelled $\mathtt{T}$, comprising the write events w_1 and w_2. Removing the $\mathtt{T}$ block (with w_1 and w_2 as non-transactional writes), this execution is deemed inconsistent, as this weak "message passing" behaviour is disallowed in the RA model. We argue that the analogous transactional behaviour in Fig. 6a must be similarly disallowed to maintain monotonicity with respect to wrapping non-transactional code in a transaction (see Theorem 3). As in SI, we cannot include the entire po in $\mathsf{rsi\text{-}hb}$ because the write-read order in transactions is not preserved by the implementation.

Similarly, $\mathsf{rsi\text{-}rf}$ is a strengthening of rf_T to account for non-transactional events: in the absence of non-transactional events $\mathsf{rsi\text{-}rf}$ reduces to $\mathsf{rf}_\mathsf{T} \cup (\mathsf{mo}; \mathsf{rf})_\mathsf{T}$ which is contained in $\mathsf{si\text{-}hb}$. The $\mathsf{rf}; [\mathcal{NT}]$ part is required to preserve the 'happens-before' relation for non-transactional code. That is, as rf is included in the hb relation of underlying memory model (RA), it is also included in $\mathsf{rsi\text{-}hb}$.

The $[\mathcal{NT}]; \mathsf{rf}; \mathsf{st}$ part asserts that in an execution where a read event r of transaction $\mathtt{T}$ reads from a non-transactional write w, the snapshot of $\mathtt{T}$ reads from w and so all events of $\mathtt{T}$ happen after w. Thus, in Fig. 6b, r' cannot read from the overwritten initialisation write to y.

For the $(\mathsf{mo}; \mathsf{rf})_\mathsf{T}$ part, consider the execution graph in Fig. 6c where there is a write event w of transaction $\mathtt{T}_1$ and a read event r of transaction $\mathtt{T}_2$ such that $(w, r) \in \mathsf{mo}; \mathsf{rf}$. Then, transaction $\mathtt{T}_2$ must acquire the read lock of $\mathtt{loc}(w)$ after $\mathtt{T}_1$ releases the writer lock, which in turn means that every event of $\mathtt{T}_1$ happens before every event of $\mathtt{T}_2$.

Remark 3. Recall that our choice of modelling SI and RSI STMs in the RA fragment is purely for presentational convenience (see Remark 2). Had we chosen a different model, the RSI consistency definition (Definition 5) would largely remain unchanged, with the exception of $\mathsf{rsi\text{-}rf} \triangleq \boxed{(\mathsf{sw}; [\mathcal{NT}]) \cup ([\mathcal{NT}]; \mathsf{sw}; \mathsf{st})} \cup \mathsf{rf}_\mathsf{T} \cup (\mathsf{mo}; \mathsf{rf})_\mathsf{T}$, where in the highlighted changes the rf relation is replaced with sw, denoting the 'synchronises-with' relation. As in the RA model $\mathsf{sw} \triangleq \mathsf{rf}$, we have inlined this in Definition 5.

SI and RSI Consistency. We next demonstrate that in the absence of non-transactional code, the definitions of SI-consistency (Proposition 1) and RSI-consistency (Definition 5) coincide. That is, for all executions G, if $G.\mathcal{NT} = \emptyset$, then G is SI-consistent if and only if G is RSI-consistent.

Theorem 2. *For all executions G, if $G.\mathcal{NT} = \emptyset$, then:*

$$G \textit{ is SI-consistent} \iff G \textit{ is RSI-consistent}$$

Proof. The full proof is given in the technical appendix [31].

Note that the above theorem implies that for all transactional programs P, if P contains no non-transactional accesses, then $\mathsf{outcomes}_{\mathrm{SI}}(P) = \mathsf{outcomes}_{\mathrm{RSI}}(P)$.

RSI Monotonicity. We next prove the *monotonicity* of RSI when wrapping non-transactional events into a transaction. That is, wrapping a block of non-transactional code inside a new transaction does not introduce additional behaviours. More concretely, given a program P, when a block of non-transactional code in P is wrapped inside a new transaction to obtain a new program P_{T}, then $\mathsf{outcomes}_{\mathrm{RSI}}(P_{\mathrm{T}}) \subseteq \mathsf{outcomes}_{\mathrm{RSI}}(P)$. This is captured in the theorem below, with its full proof given in the technical appendix [31].

Theorem 3 (Monotonicity). *Let P_{T} and P be RSI programs such that P_{T} is obtained from P by wrapping a block of non-transactional code inside a new transaction. Then:*

$$\mathsf{outcomes}_{\mathrm{RSI}}(P_{\mathrm{T}}) \subseteq \mathsf{outcomes}_{\mathrm{RSI}}(P)$$

Proof. The full proof is given in the technical appendix [31].

Lastly, we show that our RSI implementations in Sect. 2 (Figs. 3 and 5) are sound and complete with respect to Definition 5. This is captured in the theorem below. The soundness and completeness proofs are non-trivial; the full proofs are given in the technical appendix [31].

Theorem 4 (Soundness and completeness). *Let P be a program that possibly mixes transactional and non-transactional code. Let $(\!|P|\!)_{\mathrm{E}}$ denote its eager RSI implementation as given in Fig. 3 and $(\!|P|\!)_{\mathrm{L}}$ denote its lazy RSI implementation as given in Fig. 5. If for every location x and value v, every RSI-consistent execution of P contains either (i) at most one non-transactional write of v to x; or (ii) all non-transactional writes of v to x are happens-before-ordered with respect to all transactions accessing x, then:*

$$\mathsf{outcomes}_{\mathrm{RSI}}(P) = \mathsf{outcomes}_{\mathrm{RA}}((\!|P|\!)_{\mathrm{E}}) = \mathsf{outcomes}_{\mathrm{RA}}((\!|P|\!)_{\mathrm{L}})$$

Proof. The full proofs for both implementations are given in the technical appendix [31].

6 Related and Future Work

Much work has been done in formalising the semantics of weakly consistent *database transactions* [3,4,7,10–14,18,34], both operationally and declaratively. On the operational side, Berenson et al. [7] gave an operational model of SI as a multi-version concurrent algorithm. Later, Sovran et al. [34] described and operationally defined the *parallel snapshot isolation* model (PSI), as a close relative of SI with weaker guarantees.

On the declarative side, Adya et al. [3,4] introduced *dependency graphs* (similar to execution graphs of our framework in Sect. 3) for specifying transactional semantics and formalised several ANSI isolation levels. Cerone et al. [10,12] introduced *abstract executions* and formalised several isolation levels including SI and PSI. Later in [11], they used dependency graphs of Adya to develop equivalent SI and PSI semantics; recently in [13], they provided a set of algebraic laws for connecting these two declarative styles.

To facilitate client-side reasoning about the behaviour of database transactions, Gotsman et al. [18] developed a proof rule for proving invariants of client applications under a number of consistency models.

Recently, Kaki et al. [21] developed a program logic to reason about transactions under ANSI SQL isolation levels (including SI). To do this, they formulated an operational model of such programs (parametric in the isolation level). They then proved the soundness of their *logic* with respect to their proposed operational model. However, the authors did not establish the *soundness* or *completeness* of their *operational model* against existing formal semantics, e.g. [11]. The lack of the completeness result means that their proposed operational model may exclude behaviours deemed valid by the corresponding declarative models. This is a particular limitation as possibly many valid behaviours cannot be shown correct using the logic and is thus detrimental to its usability.

By contrast, transactional semantics in the STM setting with mixed (both transactional and non-transactional) accesses is under-explored on both operational and declarative sides. Recently, Dongol et al. [17] applied execution graphs [5] to specify *serialisable* STM programs under weak memory models. Raad et al. [30] formalised the semantics of PSI STMs declaratively (using execution graphs) and operationally (as lock-based reference implementations). Neither work, however, handles the semantics of SI STMs with weak isolation guarantees.

Finally, Khyzha et al. [22] formalise the sufficient conditions on STMs and their programs that together ensure strong isolation. That is, non-transactional accesses can be viewed as singleton transactions (transactions containing single instructions). However, their conditions require *serialisability* for fully transactional programs, and as such, RSI transactions do not meet their conditions. Nevertheless, we conjecture that a DRF guarantee for strong atomicity, similar to [22], may be established for RSI. That is, if all executions of a fully transactional program have no races between singleton and non-singleton transactions, then it is safe to replace all singleton transactions by non-transactional accesses.

In the future, we plan to build on the work presented here by developing 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 ones for currently unsupported models. In particular, we can reason about the SI models presented here by developing custom proof rules in the existing program logics for RA such as [24,35].

Acknowledgements. We thank the VMCAI reviewers for their constructive feedback. The first author was supported in part by a European Research Council (ERC) Consolidator Grant for the project "RustBelt", under the European Union's Horizon 2020 Framework Programme (grant agreement number 683289). The second author was supported by the Israel Science Foundation (grant number 5166651), and by Len Blavatnik and the Blavatnik Family foundation.

References

1. The Clojure Language: Refs and Transactions. http://clojure.org/refs
2. Technical specification for C++ extensions for transactional memory (2015). http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4514.pdf
3. Adya, A.: Weak consistency: a generalized theory and optimistic implementations for distributed transactions. Ph.D. thesis, MIT (1999)
4. Adya, A., Liskov, B., O'Neil, P.: Generalized isolation level definitions. In: Proceedings of the 16th International Conference on Data Engineering, pp. 67–78 (2000)
5. 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 (2014)
6. 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)
7. 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)
8. 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)
9. Blundell, C., Lewis, E.C., Martin, M.M.K.: Deconstructing transactions: the subtleties of atomicity. In: 4th Annual Workshop on Duplicating, Deconstructing, and Debunking (2005)
10. Cerone, A., Bernardi, G., Gotsman, A.: A framework for transactional consistency models with atomic visibility. In: Proceedings of the 26th International Conference on Concurrency Theory, pp. 58–71 (2015)
11. Cerone, A., Gotsman, A.: Analysing snapshot isolation. In: Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing, pp. 55–64 (2016)
12. Cerone, A., Gotsman, A., Yang, H.: Transaction chopping for parallel snapshot isolation. In: Moses, Y. (ed.) DISC 2015. LNCS, vol. 9363, pp. 388–404. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48653-5_26
13. Cerone, A., Gotsman, A., Yang, H.: Algebraic laws for weak consistency. In: CONCUR (2017)

14. Crooks, N., Pu, Y., Alvisi, L., Clement, A.: Seeing is believing: a client-centric specification of database isolation. In: Proceedings of the ACM Symposium on Principles of Distributed Computing, PODC 2017, pp. 73–82. ACM, New York (2017). https://doi.org/10.1145/3087801.3087802
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. Dias, R.J., Distefano, D., Seco, J.C., Lourenço, J.M.: Verification of snapshot isolation in transactional memory Java programs. In: Noble, J. (ed.) ECOOP 2012. LNCS, vol. 7313, pp. 640–664. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31057-7_28
17. Dongol, B., Jagadeesan, R., Riely, J.: Transactions in relaxed memory architectures. Proc. ACM Program. Lang. **2**(POPL), 18:1–18:29 (2017). https://doi.org/10.1145/3158106
18. Gotsman, A., Yang, H., Ferreira, C., Najafzadeh, M., Shapiro, M.: 'cause i'm strong enough: reasoning about consistency choices in distributed systems. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, pp. 371–384. ACM, New York (2016). https://doi.org/10.1145/2837614.2837625
19. Harris, T., Larus, J., Rajwar, R.: Transactional Memory, 2nd edn. Morgan and Claypool Publishers, San Rafael (2010)
20. 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)
21. Kaki, G., Nagar, K., Najafzadeh, M., Jagannathan, S.: Alone together: compositional reasoning and inference for weak isolation. Proc. ACM Program. Lang. **2**(POPL), 27:1–27:34 (2017). https://doi.org/10.1145/3158115
22. Khyzha, A., Attiya, H., Gotsman, A., Rinetzky, N.: Safe privatization in transactional memory. In: Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 233–245 (2018)
23. 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)
24. Lahav, O., Vafeiadis, V.: Owicki-gries reasoning for weak memory models. In: Halldórsson, M.M., Iwama, K., Kobayashi, N., Speckmann, B. (eds.) ICALP 2015. LNCS, vol. 9135, pp. 311–323. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-47666-6_25
25. Litz, H., Cheriton, D., Firoozshahian, A., Azizi, O., Stevenson, J.P.: SI-TM: reducing transactional memory abort rates through snapshot isolation. SIGPLAN Not. **49**, 383–398 (2014)
26. Litz, H., Dias, R.J., Cheriton, D.R.: Efficient correction of anomalies in snapshot isolation transactions. ACM Trans. Archit. Code Optim. **11**(4), 65:1–65:24 (2015). https://doi.org/10.1145/2693260
27. Martin, M., Blundell, C., Lewis, E.: Subtleties of transactional memory atomicity semantics. IEEE Comput. Archit. Lett. **5**(2), 17 (2006)
28. Papadimitriou, C.H.: The serializability of concurrent database updates. J. ACM **26**(4), 631–653 (1979). https://doi.org/10.1145/322154.322158
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. Raad, A., Lahav, O., Vafeiadis, V.: On parallel snapshot isolation and release/acquire consistency. In: Ahmed, A. (ed.) ESOP 2018. LNCS, vol. 10801, pp. 940–967. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89884-1_33
31. Raad, A., Lahav, O., Vafeiadis, V.: The technical appendix for this paper. https://arxiv.org/abs/1805.06196 (2018)
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. Shavit, N., Touitou, D.: Software transactional memory. In: Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing, pp. 204–213 (1995)
34. 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)
35. 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)