Decidable Verification under a Causally Consistent Shared Memory

Ori Lahav  
Tel Aviv University  
Israel  
orilahav@tau.ac.il

Udi Boker  
Interdisciplinary Center (IDC) Herzliya  
Israel  
udiboker@gmail.com

Abstract
Causal consistency is one of the most fundamental and widely used consistency models weaker than sequential consistency. In this paper, we study the verification of safety properties for finite-state concurrent programs running under a causally consistent shared memory model. We establish the decidability of this problem for the standard model of causal consistency of Burckhardt [16] (called also "causal convergence" [14] and "Strong-Release-Acquire" [27]). Our proof proceeds by developing an alternative operational semantics, based on the notion of a thread potential, that is equivalent to the existing declarative semantics and constitutes a well-structured transition system. In particular, our result allows for the verification of a large family of programs in the Release/Acquire fragment of C/C++11 (RA). Indeed, while verification under RA was recently shown to be undecidable for general programs, since RA coincides with the model we study here for write/write-race-free programs, the decidability of verification under RA for this widely used class of programs follows from our result. The novel operational semantics may also be of independent use in the investigation of weakly consistent shared memory models and their verification.

ACM Reference Format:

1 Introduction
Suppose that one wants to verify that a given sequential program satisfies a certain safety specification (e.g., that it never crashes). If the data domain is bounded, we can represent the program as a finite-state transition system, and this verification problem is trivially decidable. Moving to concurrent programs, assuming (non-realistic) sequentially consistent shared memory semantics, does not change much—the memory constitutes another finite-state system, and its synchronization with the interleaving of the systems representing the different threads is easily expressible as a finite-state system as well. On the other hand, if the memory does not ensure sequential consistency, but rather provides weaker consistency guarantees, the decidability of the safety verification problem is completely unclear.

In this paper, we are interested in the safety verification problem under causally consistent shared memory. Causal consistency, which appears in several different forms, is one of the most fundamental consistency models weaker than sequential consistency. It is especially common and well studied in distributed databases (see, e.g., [34] and the MongoDB documentation [37]). Roughly speaking, by allowing nodes to disagree on the relative order of some memory operations, and require global consensus only on the order of "causally related" operations, causal consistency allows scalable, partition-tolerant and available implementations.

Nowadays, causal consistency models have become central also in multithreaded programming. In particular, the Release/Acquire model (RA) is a form of causal consistency that specifies the semantics of C/C++11 for synchronization accesses annotated with memory_order_release and memory_order_acquire [12, 21, 22]. A stronger form of causal consistency, called SRA (for Strong Release/Acquire), which is equivalent to the standard causal consistency model in distributed databases [16], characterizes the guarantees provided by “multi-copy atomic” multiprocessor architectures, such as POWER. Specifically, as shown in [27], SRA precisely captures the guarantees provided by the POWER architecture for programs compiled from the C/C++’s release/acquire fragment.

Despite its centrality, until recently not much was known about the safety verification problem under causal consistency. The challenge arises first since the standard semantics of causal consistency models is declarative (identifying program behaviors with partially ordered execution histories that obey certain formal consistency constraints), while verification is typically applied on operational models. Moreover, operational versions of causal consistency are inherently infinite-state, as threads may generally read from an unbounded past. In fact, the reduction of Atig et al. [9] from reachability in lossy FIFO channel machines to safety verification under x86-TSO semantics can be straightforwardly adapted to causally consistent models (specifically, RA and

[1]This equivalence excludes read-modify-writes, which are crucial in multithreaded programming but do not exist as primitives in distributed databases.
SRA). This implies a non-primitive recursive lower bound on
the safety verification problem under causal consistency.
Very recently, Abdulla et al. [2] proved that the safety verifica-
tion problem is undecidable under one instance of causal
consistency, namely the the RA model.

Our main contribution in this paper is to establish the
decidability of safety verification under the SRA model. If
one is specifically interested in verification under RA, our
result provides an under-approximation (a bug under SRA
implies a bug under RA), and, since RA and SRA coincide on
write/write-race-free programs, we obtain the decidability of
safety verification under RA for this large and widely used
class of programs.

To obtain decidability, we develop a novel operational se-
manitics for programs running under SRA that is equivalent
to their declarative semantics. The semantics is infinite-state,
but as we show, can be utilized to decide the safety verifica-
tion problem by using the framework of well-structured
transition systems [1, 5, 20]. The key idea in this semantics
is to maintain the potential of future reads of each thread
in the machine state. Thus, read transitions are very sim-
ple, they only consume a prefix of the potential, and the
complexity is left for write transitions that need to properly
increase the potentials of the different threads in a way that
ensures causal consistency. Our fundamental observation is
that the way the potential of a certain thread increases when
another thread writes to memory can be defined solely in
terms of the existing potentials of the two threads. Centered
about potential, this novel semantics can be made “lossy”,
making it a suitable well-structured transition system. In-
deed, losing some parts of the possible potential never allows
for additional behaviors. This intuition is made precise in
our formalized correspondence proofs, which establish sim-
ulations (forward for one direction and backward for the
converse) between this lossy semantics and the straightforward “operationalization” of SRA’s declarative semantics.

Related Work. Causally consistent shared memory models,
their verification problems and approaches to address these
problems were recently outlined in [26], where the problem
we resolve is left open. As mentioned above, Abdulla et al.
[2] proved that safety verification under RA is undecidable.
Operational “message-passing” semantics for SRA was de-
veloped in [27]. It is inadequate for our purposes as it cannot
be made “lossy” without affecting its allowed outcomes.

The safety verification problem was previously investi-
gated under TSO—the “total store ordering” model of x86
multiprocessors, which, being multi-copy-atomic, is stronger
than any of the models studied here. Atig et al. [9, 10] es-
ablish the decidability of this problem (and the non-primitive
recursive lower bound) by reducing it to (and from) reach-
ability in lossy channel systems. Since causal consistency
models are not multi-copy atomic and they lack any notion
of a global mapping from locations to values, the idea be-
hind their reduction cannot be applied for SRA. Notably,
SRA cannot be fully explained by program transformations
(instruction reordering and merging) [30], whereas, with the
exception of the recent undecidability in [2], all existing
results (of [10] in particular) are for models that are fully
accounted for by such transformations.

More recently, Abdulla et al. [3] greatly simplified previous
proofs for TSO (and demonstrated much better practical
running times on certain benchmarks) by developing and
utilizing a “load-buffer” semantics for TSO. Load-buffers are
roughly similar to our potential lists, but while load buffers
are FIFO queues, our lists necessarily allow the insertion of
future reads at different positions, subject to certain (novel)
conditions ensuring that causal consistency is not violated.
In addition, the “load-buffer” semantics for TSO includes
a global machine memory, while our semantics does not
employ any such notion.

Verification of programs under causal consistency (espe-
ially under RA) has received considerable amount of atten-
tion in recent years. The different approaches include (non-
automated) program logics [19, 23, 29, 45, 46], (bounded)
model checking based on partial order reduction [2, 4, 24, 32]
and robustness verification [15, 28, 38]. The latter approach
reduces the verification problem to the verification under
sequential consistency and the verification of the program’s
robustness against causal consistency. Thus, this approach
cannot work for programs that meet their safety specification
but still exhibit non-sequentially-consistent behaviors.

Finally, the problem asking whether a given implementa-
tion provides causal consistency guarantees was studied in
[14]. It is, however, independent from verification of client
programs assuming causal consistency, as we study here.

Outline. The rest of this paper is organized as follows. In §2
we provide preliminary definitions. In §3 we present the SRA
model and its safety verification problem, and prove that
RA and SRA coincide for write/write-race-free programs. In
§4 we present straightforward operational version of SRA’s
declarative semantics. In §5 we introduce our novel opera-
tional semantics of SRA. In §6 we show how this semantics is
used to decide the safety verification problem. We conclude
in §7. The appendices provide additional examples (Appen-
dix A), and full proofs (Appendices B to D). A mechanized
proof of the equivalence of the two semantics of SRA in the
Coq proof assistant is available upon request.

2 Preliminaries

SRA is a declarative memory model, defined by imposing
certain consistency constraints on execution graphs. The latter
describe the (partially ordered) history of a program run. In
this section, we provide the preliminaries for declarative
memory model: We introduce a toy programming language
(§2.1), interpret its programs as transition systems (§2.2) and
associate these transition systems with declarative execution graphs (§2.3).

### 2.1 Programming Language

Let $\text{Val} \subseteq \mathbb{N}$, $\text{Loc} \subseteq \{x, y, \ldots\}$, $\text{Reg} \subseteq \{a, b, \ldots\}$ be finite sets of values, (shared) memory locations, and register names. Figure 1 presents our toy language. Its expressions are constructed from registers (local variables) and values. Instructions include assignments and conditional branching, as well as memory operations. Intuitively speaking, an assignment $r := e$ assigns the value of $e$ to register $r$ (involving no memory access); if $e$ goto $n$ sets the program counter to $n$ iff the value of $e$ is not 0; a “write” $x := e$ stores the value of $e$ in $x$; a “read” $r := x$ loads the value of $x$ to register $r$; $r := \text{FADD}(x, e)$ atomically increments $x$ by the value of $e$ and loads the old value of $x$ to $r$; $r := \text{XCHG}(x, e)$ atomically swaps $x$ to the value of $e$ and loads the old value of $x$ to $r$; and $r := \text{CAS}(x, e_0, e_1)$ atomically loads the value of $x$ to $r$, compares it to the value of $e_0$, and if the two values are equal, replaces the value of $x$ by the value of $e_1$.

A sequential program $S$ is a function from a set of the form $\{0, 1, \ldots, N\}$ (the possible values of the program counter) to instructions. We denote by $\text{SProg}$ the set of all sequential programs. A (concurrent) program $P$ is a top-level parallel composition of sequential programs, defined as a mapping from a finite set $\text{Tid} \subseteq \{T_1, T_2, \ldots\}$ of thread identifiers to $\text{SProg}$. In our examples, we often write sequential programs as sequences of instructions delimited by line breaks, use ‘|’ for parallel composition, and refer to the program threads as $T_1, T_2, \ldots$ following their left-to-right order in the program listing (see, e.g., Ex. 3.5 on Page 8).

### 2.2 From Programs to Labeled Transition Systems

Sequential and concurrent programs induce labeled transition systems.

#### Labeled transition systems.

A labeled transition system (LTS) $\mathcal{A}$ over an alphabet $\Sigma$ is a triple $(Q, \mathcal{Q}_0, T)$, where $Q$ is a set of states, $\mathcal{Q}_0 \subseteq Q$ is the set of initial states, and $T \subseteq Q \times \Sigma \times Q$ is a set of transitions. We denote by $\mathcal{A}.Q$, $\mathcal{A}.q_0$ and $\mathcal{A}.T$ the components of an LTS $\mathcal{A}$; write $\mathcal{A}.q \xrightarrow{\sigma} \mathcal{A}.q'$ for the relation $\{ (q, q') \mid (q, \sigma, q') \in \mathcal{A}.T \}$ and $\mathcal{A}.q \xrightarrow{\sigma} \mathcal{A}.q'$ for $\bigcup_{\mathcal{A}.q \in \mathcal{A}.Q} \mathcal{A}.q \xrightarrow{\sigma} \mathcal{A}.q'$. A state $q \in \mathcal{A}.Q$ is reachable in $\mathcal{A}$ if $q_0 \xrightarrow{\sigma} \mathcal{A}.q$, for some $q_0 \in \mathcal{A}.Q_0$. A sequence $\sigma_1, \ldots, \sigma_n$ is a trace of $\mathcal{A}$ if $q_0 \xrightarrow{\sigma_1} \cdots \mathcal{A}.q \xrightarrow{\sigma_n} \mathcal{A}.q'$ for some $q_0 \in \mathcal{A}.Q_0$ and $q \in \mathcal{A}.Q$. The set of predecessors of a set $S \subseteq \mathcal{A}.Q$ w.r.t. a symbol $\sigma \in \Sigma$, denoted by $\text{pred}_S(\sigma)$, is given by $\{ q \in \mathcal{A}.Q \mid \exists q' \in S. q \xrightarrow{\sigma} q' \}$. We define $\text{pred}_S(S) \equiv \bigcup_{\sigma \in \Sigma} \text{pred}_S(\sigma)$.

For sequential programs the alphabet is the set of labels (extended with $\epsilon$ for silent transitions), as defined next.

#### Definition 2.1.

A label is either $\text{R}(x, q_0)$ (read label), $\text{W}(x, q_0)$ (write label) or $\text{RMW}(x, q_0, q_w)$ (read-modify-write label), where

$x \in \text{Loc}$ and $q_0, q_w \in \text{Val}$. We denote by $\text{Lab}$ the set of all labels. The functions $\text{typ}, \text{loc}, \text{val}_R$, and $\text{val}_W$ return (when applicable) the type $(R/W/\text{RMW})$, location, read value and written value of a given label $l$.

A sequential program $S \in \text{SProg}$ induces an LTS over $\text{Lab} \cup \{\epsilon\}$. Its states are pairs $s = \langle pc, \phi \rangle$ where $pc \in \mathbb{N}$ (called program counter) and $\phi : \text{Reg} \rightarrow \text{Val}$ (called local store, and extended to expressions in the obvious way). Its only initial state is $\langle 0, \lambda r \in \text{Reg}, 0 \rangle$ and its transitions are given in Fig. 2, following the informal description above. (In particular, a read instruction in $S$ induces $\text{Val}$ transitions with different read labels.) We identify sequential programs with their induced LTSs (when writing, e.g., $\text{SProg}$ and $\text{SProg}.$)

In turn, a concurrent program $P$ is identified with an LTS over $\text{Tid} \times (\text{Lab} \cup \{\epsilon\})$. Its states are functions, often denoted by $\mathcal{P}$, assigning a state in $P(r).Q$ to every $r \in \text{Tid}$; its initial states set is $\{ \mathcal{P} \mid \forall r. \mathcal{P}(r) \in P(r).Q_0 \}$; and its transitions are “interleaved transitions” of $P$’s components, given by:

$$
\begin{align*}
\text{Lab} & \xrightarrow{\mathcal{A}} \text{Lab} & \text{Lab} & \xrightarrow{\mathcal{A}} \text{Lab} \\
\mathcal{A} & \xrightarrow{\mathcal{Tid} \times \text{Lab}} \mathcal{A} & \mathcal{A} & \xrightarrow{\mathcal{Tid} \times \text{Lab}} \mathcal{A}
\end{align*}
$$

### 2.3 From LTSs to Execution Graphs

We present the general notions used to assign declarative semantics to concurrent programs. First, we define execution graphs, starting with their nodes, called events.

#### Definition 2.2.

An event is a triple $e = \langle r, n, l \rangle$, where $r \in \text{Tid}$ is a thread identifier, $n \in \mathbb{N}$ is a serial number and $l \in \text{Lab}$ is a label (Def. 2.1). The function $\text{tid}$ returns the thread identifier of an event. The functions $\text{typ}, \text{loc}, \text{val}_R$, and $\text{val}_W$ are lifted to events in the obvious way. We denote by $E$ the set of all events, and use $R, W, \text{RMW}$ for its subsets: $R \equiv \{ e \mid \text{typ}(e) \in \{R, \text{RMW}\} \}$, $W \equiv \{ e \mid \text{typ}(e) \in \{W, \text{RMW}\} \}$ and $\text{RMW} \cap \mathbb{R} \cap \mathbb{W}$. Sub/superscripts are used to restrict these sets to certain location (e.g., $\text{Loc}_R = \{ w \in \mathbb{W} \mid \text{loc}(w) = x \}$) and/or thread identifier (e.g., $E^r = \{ e \in E \mid \text{tid}(e) = r \}$).

Our representation of events induces a partial order on them: events of the same thread are ordered according to their serial numbers (i.e., $(r_1, n_1, l_1) < (r_2, n_2, l_2)$ iff $r_1 = r_2$ and $n_1 < n_2$). In turn, an execution graph consists of a set of events, a reads-from mapping that determines the write event from which each read event reads its value, and a modification order that totally orders the writes to each location.

#### Definition 2.3.

A relation $\mathcal{R}$ is a reads-from relation for a set $E$ of events if the following hold:

- If $\langle w, r \rangle \in \mathcal{R}$, then $w \in E \cap \mathbb{W}, r \in E \cap \mathbb{R}$, $\text{loc}(w) = \text{loc}(r)$ and $\text{val}_W(w) = \text{val}_W(r)$.
- If $\langle w_1, r \rangle, \langle w_2, r \rangle \in \mathcal{R}$, then $w_1 = w_2$ (that is, $\mathcal{R}^{-1} = \{ (r, w) \mid (w, r) \in \mathcal{R} \}$ is functional).
- $\forall r \in \mathbb{R} \exists w. \langle w, r \rangle \in \mathcal{R}$ (each read event reads from some write event).
where\( e := r \mid v \mid e + e \mid e = e \mid e \neq e \mid ... \)\( \phi \). For a set \( \mathcal{E} \), \( \phi \) denotes its domain; \( \mathcal{R} \) and \( \mathcal{R}^{-1} \) denote its reflexive and transitive closures; and \( \mathcal{R}^{-1} \) denotes its inverse. (The left) composition of relations \( R_1 \) and \( R_2 \) is denoted by \( R_1 \circ R_2 \). We denote by \( [A] \) the identity relation on a set \( A \), and so \([A] \circ [B] = R \cap (A \times B)\). Causal consistency models are based on the following basic derived “happens-before” relation:

\[
\text{G.hb} \triangleq (\text{G.po} \cup \text{G.rf})^+
\]

The happens-before relation captures the “causality” relation in execution graphs. In words, \( \text{hb} \) is the smallest transitive relation that contains the program order (po) and the reads-from (rf) relations. We note that all reads synchronize with the writes they read from (rf \( \subseteq \text{hb} \)), in contrast to more elaborate models like RC11 [31], where only certain reads-from edges induce synchronization.

Given \( \text{hb} \), the SRA model consists of three constraints, each of which forbids a certain pattern in execution graphs. The three disallowed patterns are illustrated as follows:

\[
\text{irr-hb-mo}. \text{ This constraint requires that the modification order mo “agrees” with the causality order:}
\]

\[
(G.hb \cup G.mo)^+ \text{ is irreflexive} \quad \text{(irr-hb-mo)}
\]

In particular, it implies that \( G.hb \) is indeed a partial order.

Thus, SRA forbids so-called “load-buffering” behaviors [36],

\[
\text{Figure 1. Domains, metavariabes and programming language syntax.}
\]

\[
\text{Figure 2. Transitions of LTS induced by a sequential program S in SProg.}
\]
which, unless restricted appropriately, lead to the infamous "out-of-thin-air" problem [11].

**read-coherence.** This constraint intuitively requires that "a thread cannot read a value when it is aware of a later value written to the same location". Identifying "thread 𝜏 being aware of some write event 𝑤" with an hb-path from 𝑤 to (some event of) 𝜏, and using he modification order mo to interpret one write being "later" than another, the precise condition requires that:

\[ G \cdot \text{mo} ; G \cdot \text{hb} ; G \cdot r f^{-1} \text{ is reflexive } \quad \text{(read-coherence)} \]

Indeed, if a read event 𝑟 reads from a write event 𝑤₁, while being aware of an mo-later write event 𝑤₂ to the same location, we have \( ⟨𝑤₁, 𝑤₂⟩ \in \text{mo} \), \( ⟨𝑤₂, 𝑟⟩ \in \text{hb} \) and \( ⟨𝑟, 𝑤₁⟩ \in r f^{-1} \).

**atomicity.** A final condition ensures that RMWs are stronger than a read followed by a write. It requires that RMWs read from their immediate mo-predecessors:

\[ G \cdot \text{mo} ; G \cdot \text{mo} ; G \cdot r f^{-1} \text{ is reflexive } \quad \text{(atomicity)} \]

In words, if an RMW event 𝑒 is reading from a write event 𝑤, then no write event can intervene mo-between 𝑤 and 𝑒.

We refer to execution graphs that meet the three conditions above as SRA-consistent. With this definition, we can formally present the reachability problem under SRA, which we prove to be decidable in this paper.

**Definition 3.2.** We call a state \( \bar{p} \) of a program \( P \) reachable under SRA if some SRA-consistent execution graph is generated by \( P \) with final state \( \bar{p} \).

**Definition 3.3 (SRA Reachability).** The reachability problem under SRA is given by:

**Input:** a program \( P \) and a "bad" state \( \bar{p} \in P.Q. \)

**Question:** is \( \bar{p} \) reachable under SRA?

A lower complexity bound to this problem is achieved by reduction from reachability in lossy FIFO channel machines, straightforwardly following the analogous reduction of Atig et al. [9] to safety verification under x86-TSO.

**Theorem 3.4.** SRA reachability is non-primitive-recursive.

### 3.1 Other Formulations of SRA

Our presentation above of SRA follows C/C++11’s formalization [12, 31], where the RA model is the fragment of the C/C++11 model consisting of release stores, acquire reads and acquire-release RMWs, and SRA is a strengthening of RA proposed in [27]. In addition, SRA appears (in multiple disguises) in the literature:

**POWER.** As proved in [27], SRA precisely coincides with the POWER model of [7], when the latter is restricted to programs that result from compiling C/C++11 programs in the release/acquire fragment, using the standard compilation scheme [35] (that is, placing 1wsync before every store and ctrl+isync after every load).

**Distributed Key-Value Stores.** Ignoring RMWs, the SRA model is equivalent to the causal convergence model, denoted by CCv, of [14] (when applied to the standard read/write memory sequential specification), as well as to the causal consistency model of [34] when restricted to single-instruction transactions. These models are formulated in [16, 18] in terms of visibility (ois) and arbitration (ar) relations. One direction of the correspondence follows by setting ois = hb and taking ar to be some total order extending hb \( \cup \text{mo} \). For the converse, one takes rf to relate each read 𝑟 with the ar-maximal write to the same location that is ois-before 𝑟, and sets mo = \( \bigcup_{x \in \text{loc}} [W_x] ; ar ; [W_x] \). Furthermore, our program order (po) corresponds to session order (so), and SRA’s consistency ensures strong session guarantees (so \( \subseteq \text{ois} \)) [44]. Intended for distributed systems, these models do not provide RMWs. Nevertheless, when a location is only accessed by RMWs, its accesses are totally ordered by hb, which corresponds to marking of certain transactions as serializable, as in the Red-Blue model of [33] (see also [13]).

**Parallel-Snapshot-Isolation.** When all store instructions are implemented using atomic exchanges (implementing \( x := e \) as \( r := \text{XCHG}(x, e) \)), SRA precisely captures the parallel snapshot isolation model (PSI) [8, 13, 17, 42] when restricted to single-instruction transactions. Hence, our decidability result for SRA entails the decidability for PSI with single-instruction transactions.

### 3.2 Examples

We list some examples to demonstrate SRA (some of which are revisited in the sequel). Most of the examples are well-known litmus tests (where initialization, which is not implicitly included in our execution graphs, has been made explicit). To simplify the presentation, instead of referring to reachable program states, we consider possible program outcomes assigning final values to (some) registers. An outcome \( O \) is allowed for a program under a declarative model \( X \) if some state in which the registers are assigned their values in \( O \) is reachable under \( X \) (see Def. 3.2). We use program comment annotations ("//") to denote particular outcomes.

**Example 3.5 (Store buffering).** The following program outcome is allowed by SRA.

```
\[
\begin{align*}
\text{x := 0} & \quad \text{y := 0} \\
\text{x := 1} & \quad \text{y := 1} \\
\text{a := y} / / & \quad \text{b := x} / /\\
\text{\checkmark SRA} & \quad \text{\checkmark SRA}
\end{align*}
\]
```

In its execution graph the rf-edges are forced because of the read values, whereas the mo-edges are forced due to write-coherence and irr-hb-mo. It can be easily verified that the execution graph is SRA-consistent.

**Example 3.6 (Message passing).** SRA supports the very common "flag-based" synchronization. That is, the following
An execution graph for this outcome must have \( rf \) and \( mo \)-edges as depicted above. However, we have \( mo \)'s from \( W(x, 0) \) to \( W(x, 1) \), \( hb \) from \( W(x, 1) \) to \( R(x, 0) \) and \( rf \) from \( W(x, 0) \) to \( R(x, 0) \). Hence, read-coherence does not hold, and the execution graph is not SRA-consistent.

**Example 3.7** (Independent reads of independent writes). A main difference between SRA and the x86-TSO model [39] is that the former is non-multi-copy-atomic. Namely, different threads may observe different stores in different orders. Thus, unlike x86-TSO, the SRA model allows the following outcome, in which \( T_2 \) observes \( W(x, 1) \) but not \( W(y, 1) \), while \( T_3 \) observes \( W(y, 1) \) but not \( W(x, 1) \).

\[
\begin{align*}
x &:= 0 \quad a := x \neq 1 \quad c := y \neq 1 \quad y := 0 \; \checkmark \text{SRA} \\
x &:= 1 \quad b := y \neq 0 \quad d := x \neq 0 \quad y := 1
\end{align*}
\]

**Example 3.8.** For the implementation of locks, it is crucial that two RMWs never read from the same write:

\[
\begin{align*}
x &:= 0 \quad a := \text{CAS}(x, 0, 1) \neq 0 \\
b := \text{CAS}(x, 0, 1) \neq 0
\end{align*}
\]

Since \( mo \) must order the two RMWs and \( irr-hb-mo \) dictates that \( mo : rf \) is irreflexive, any order of the RMWs entails a violation of atomicity.

### 3.3 Relation to the RA Model

The RA model is weaker than SRA. It imposes read-coherence and atomicity, just like SRA, but instead of \( irr-hb-mo \), it only disallows the following patterns:

\[
\begin{align*}
E &\xrightarrow{hb} \quad \text{irr-hb} & W_X \\
W_X &\xrightarrow{mo} \quad \text{write-coherence}
\end{align*}
\]

First, \( irr-hb \) requires \( hb \) to be a partial order:

\[ G.hb \text{ is irreflexive} \quad \text{(irr-hb)} \]

Second, instead of a global agreement between \( mo \) and \( hb \), RA only requires a local agreement:

\[ G.mo; G.hb \text{ is irreflexive} \quad \text{(write-coherence)} \]

In words, if \( hb \) orders two writes to the same location, then \( mo \) must follow the same order.

**Example 3.9.** Cycles in \( hb \cup mo \) involving only one location are disallowed by write-coherence (using the fact that \( mo \) is total on writes to the same location). In contrast, \( irr-hb-mo \) (in SRA) restricts the relation between \( [W_x]; [W_y] \text{ and } [W_z] \) when \( x \neq y \). The following example (adapted from [47]) demonstrates the difference:

\[
\begin{align*}
x &:= 1 \quad y := 2 \\
a := y \neq 1 \quad a := x \neq 1
\end{align*}
\]

An execution graph for this outcome must have \( rf \) and \( mo \)-edges as depicted above (to satisfy read-coherence), and it contains a \( (hb \cup mo) \)-cycle, which is allowed by RA and disallowed by SRA.

Since \( irr-hb-mo \) implies both \( irr-hb \) and write-coherence, the following trivially holds:

**Proposition 3.10.** SRA-consistency implies RA-consistency.

Reachability under RA is defined analogously to reachability under SRA (replacing “SRA” with “RA” in Def. 3.2). Then, we clearly have that all states of a program \( P \) that are reachable under SRA are also reachable under RA. The converse does not hold in general, but it does hold for the large and widely used class of write/write-race-free programs. Inspired by DRF models [6], we show that write/write-race freedom of SRA-consistent execution graphs suffices, so that programmers may adhere to a safe programming discipline without even understanding RA. The proof is given in Appendix B.

**Definition 3.11.** An execution graph \( G \) is write/write-race free if for every \( w_1, w_2 \in G \cdot W \) with \( 1oc(w_1) = 1oc(w_2) \), we have \( w_1 = w_2, \{ w_1, w_2 \} \in E.hb \) or \( (w_2, w_1) \in G.hb \). A program \( P \) is write/write-race free (under SRA) if every SRA-consistent execution graph that is generated by \( P \) (with some final state) is write/write-race free.

**Theorem 3.12.** Let \( P \) be a program that is write/write-race free under SRA. Then, the sets of states of \( P \) that are reachable under SRA and RA coincide.

### 4 Operationalizing the SRA Model

In this section, we present an operational semantics for SRA, formulating it as a memory system. While the formulation in §3 is declarative, it is straightforward to “operationalize” it. Indeed, instead of first generating a program execution graph and then checking for SRA-consistency, one may impose consistency at each step of an incremental construction of the execution graph. This results in an equivalent operational presentation, which is arguably simpler and easier to relate to the alternative semantics we define in §5.

**Definition 4.1.** A memory system is a (possibly infinite) LTS over the alphabet \((Tid \times Lab) \cup \{\epsilon\} \)
The alphabet symbols of the memory system are pairs in Tid × Lab, representing the thread identifier and the label of the operation, or ε for internal (silent) memory actions.

By synchronizing a program and a memory system, we obtain a concurrent system:

**Definition 4.2.** A program $P$ and a memory system $M$ form a concurrent system, denoted by $P_M$. It is an LTS over $(\text{Tid} \times (\text{Lab} \cup \{\varepsilon\})) \cup \{\varepsilon\}$ whose set of states is $P \times Q \times M;Q$; its initial states set is $P_0 \times Q_0 \times M_0$; and its transitions are "synchronized transitions" of $P$ and $M$, given by:

$$\begin{align*}
\forall l \in \text{Lab} & \quad (p,m) \xrightarrow{\tau,l} (p',m') \\
\forall p \in P & \quad (p,m) \xrightarrow{\tau,c} (p',m') \\
\forall m \xrightarrow{\tau} m' & \quad (p,m) \xrightarrow{\tau} (p,m')
\end{align*}$$

Next, we present the memory system $\text{opSRA}$ that is equivalent to SRA (in the sense that is made formal in Thm. 4.4). We also refer the reader to Fig. 5 on Page 14, which illustrates a run of SRA for the SB example.

The states of $\text{opSRA}$ are execution graphs capturing (partially ordered) histories of executed actions ($\text{opSRA}_Q \triangleq \text{EGraph}$); the (only) initial state is the empty execution graph $\emptyset \text{opSRA}_Q \triangleq \{\emptyset\}$; and the transitions are given in Fig. 3. A write step by thread $\tau$ adds a corresponding fresh write event $\varepsilon$ to the graph placed after all events of thread $\tau$ and extends $\text{mo}$ to order $\varepsilon$ after all existing writes to the same location. A read step by thread $\tau$ adds a corresponding fresh read event and justifies it with a reads-from edge. Its source $w$ must be a write event to the same location ($w \in G.W_x$), writing the value being read ($\text{val}(w) = v$), and the thread executing the read is not aware of an $\text{mo}$-later write to the same location ($w \notin \text{dom}(G.\text{mo};G.\text{hb}^+;[E'])$). An $\text{RMW}$ step combines a $\text{READ}$ and a $\text{WRITE}$, but it is enforced to pick the $\text{mo}$-maximal write to the relevant location in the current graph as the reads-from source of the freshly added $\text{RMW}$.

This semantics exploits the fact that $\text{hb} \cup \text{mo}$ is acyclic in SRA-consistent execution graphs (as per $\text{irr-hb-mo}$). Hence, to generate an SRA-consistent execution graph in a run of an operational semantics, we can follow a total order extending $\text{hb} \cup \text{mo}$, which guarantees that writes are executed following their $\text{mo}$-order. In turn, since $\text{RMWs}$ should read from their immediate $\text{mo}$-predecessor, we require that $\text{RMWs}$ read from the current $\text{mo}$-maximal write.

The next definition and simple theorem formalize the correspondence between SRA and opSRA.

**Definition 4.3.** A state $\bar{p}$ of a program $P$ is reachable under a memory system $M$ if $(\bar{p}, m)$ is reachable in $P_M$ for some $m \in M.Q$.

**Theorem 4.4.** A state $\bar{p}$ of program $P$ is reachable under SRA (see Def. 3.2) iff it is reachable under opSRA.

*Proof.* Given an SRA-consistent execution graph $G$, one obtains a run of SRA by following any total order extending $G.\text{hb} \cup G.\text{mo}$. The preconditions required by each step follow directly from the fact that $G$ is SRA-consistent. For the converse, it suffices to note that all reachable states of opSRA are SRA-consistent execution graphs. Hence, if $(\bar{p}, G)$ is reachable in opSRA, then $G$ is an SRA-consistent execution graph that is generated by $P$ with final state $\bar{p}$. □

## 5 Making Strong Release/Acquire Lossy

For resolving the reachability problem under SRA, we introduce an alternative memory system, which we call $\text{IoSRA}$ (for "lossy-SRA"). In this section, we present $\text{IoSRA}$, establish its equivalence to opSRA, and show how it is used to decide the reachability problem. We begin with an intuitive discussion to motivate our definitions.

A memory state of $\text{IoSRA}$ maintains a collection of "read-option" lists for each thread, called the potential of the thread, where each read option $\omega$ contains a location $1\text{oc}(\omega)$, a value $\text{val}(\omega)$ and two other components that are explained below. Each read-option list stands for a sequence of possible future reads of the thread, listing the writes that it may read in the order that it may read them. For example, the list $\omega_1 \cdot \omega_2$ allows the thread to read $\text{val}(\omega_1)$ from location $1\text{oc}(\omega_1)$ and then $\text{val}(\omega_2)$ from location $1\text{oc}(\omega_2)$. These lists do not ascribe mandatory continuations, but rather possible futures (hence, read options). In the beginning, the empty list is assigned to all threads—before any write is executed, no reads are possible. In addition, the semantics is designed so that read-option lists are "lossy", allowing a non-deterministic step that removes arbitrary options from the lists.

The read-option lists in the potentials dictate the possible read steps threads can take: for a thread $\tau$ to read $\omega$ from $x$, an option $\omega$ with $\text{val}(\omega) = v$ and $1\text{oc}(\omega) = x$ must be the first in each of $\tau$’s lists. In turn, the step consumes these options, discarding the first element from each of $\tau$’s lists.

A write step is more involved, encapsulating the requirements of opSRA. First, since opSRA performs write events following their $\text{mo}$-order, when a thread writes to $x$, it cannot later read $x$ from a write that was already performed (this would violate read-coherence in terms of SRA). Accordingly, we do not allow a thread to write to $x$ if some read option $\omega$ with $1\text{oc}(\omega) = x$ appears in its potential. Second, when a thread performs a write of $v$ to $x$, it allows future reads from this write. That is, read options $\omega$ with $1\text{oc}(\omega) = x$ and $\text{val}(\omega) = v$ may be added to every list of every thread. But, where in the lists should we allow to add such options? The following examples demonstrate two possible cases. We write in them $\omega_x^* \omega_y$ for a read option of value $v$ from location $x$.

**Example 5.1.** Consider the IRIW program with its (SRA-allowed) outcome in Ex. 3.7. Clearly, the first step may only be a write by $T_1$ or $T_4$. Suppose, w.l.o.g., that $T_1$ begins. Since $T_3$ reads 0 from $x$, a read option $\omega_1^* \omega_2$ should be added in the lists of $T_3$. Now, before reading 0 from $x$, $T_3$ has to read 1...
from y. Hence, when T4 writes 1 to y, a read option o_y^0 should be placed before o_x^0 in the lists of T3.

**Example 5.2.** Consider the MP program with its outcome in Ex. 3.6. It is forbidden under SRA, and so we need to avoid the following scenario: First, T1 writes 0 to x and adds a corresponding option o_x^0 to the (initially empty) list of T2, and then writes 1 to x without adding any option to any list (no thread reads 1 from x in this program outcome). Then, T1 further writes 1 to y and adds a corresponding option o_y^0 in the list of T1 placed before o_x^0. Finally, T2 may run: read 1 from y (consuming o_y^0) and then 0 from x (consuming o_x^0).

The restriction we impose on the positions of the added read options stems from the following key observation:

**Shared-memory causality principle:** After thread π reads from a certain write executed by thread τ, it can perform a sequence of operations only if thread τ could perform the same sequence immediately after it executed the write.

Indeed, if thread τ has just performed a write w, then after thread π reads from w, it “synchronizes” with τ and it is thus confined by the sequences of reads that τ may perform. Hence, to allow the addition of a read option o in certain positions of a list L of some thread π, we require a justification: the suffix of L after the first occurrence of o should be a subsequence of a read-option list of the writing thread τ. This guarantees that after π reads from a write w of τ, it will not be able to read something that τ could not read at the time that it wrote w. (Revisiting Ex. 5.2, the read option o_y^0 cannot be placed before o_x^0, because T1 cannot have o_x^0 in its lists at the point of writing 1 to y.)

Now, since the potential of thread τ is used both for 1. dictating future reads of τ, and 2. justifying placement of read options that are generated by τ’s write steps, we may need more than one option list for each thread. We also allow to discard existing lists in silent moves of the memory system. This is demonstrated in the following example.

---

**Example 5.3.** Consider the following program, whose annotated outcome is allowed under SRA:

\[
\begin{align*}
x &:= 0 \\
y &:= 0 \\
a &:= z \not\in \{0, 1\} \\
b &:= x \not\in \{0, 1\} \\
c &:= w \not\in \{0, 1\} \\
d &:= y \not\in \{0, 1\} \\
z &:= 1
\end{align*}
\]

Suppose that it can be obtained by the memory system outlined above with one read-option list per thread (i.e., singleton potentials). Suppose, w.l.o.g., that z := 1 is the last write performed in the execution. Later, T3 has to read 1 from y and 0 from x. Hence, its read-option list must include o_x^0 and o_y^0 in this order. In addition, a read option o_z^0 should be placed in T3’s list before o_x^0 · o_y^0. The justification for it requires o_x^0 · o_y^0 to be a subsequence of T3’s list. This implies that T3’s list should contain some interleaving of o_x^0 · o_y^0 and o_x^0 · o_y^0. But, no such interleaving is a possible future for T3 (and thus cannot be generated by loSRA): reading o_x^0 does not allow to read o_y^0 later; and reading o_x^0 does not allow to read o_y^0 later. By allowing more than one read-option list per thread, we can have o_x^0 · o_y^0 and o_x^0 · o_y^0 in two separate lists in the potential of T3—both are possible continuations for it after z := 1. Then, after executing z := 1, T3 may “lose” the justifying list o_x^0 · o_y^0, and choose to continue with o_x^0 · o_y^0 for its own reads.

Another complication arises due to the fact that read options do not uniquely identify write events in the execution graph (this is unavoidable: for the decision procedure, we need the alphabet of read options to be finite):

**Example 5.4.** Consider the following program:

\[
\begin{align*}
x &:= 0 \\
y &:= 0 \\
a &:= z \not\in \{0, 1\} \\
b &:= x \not\in \{0, 1\} \\
c &:= w \not\in \{0, 1\} \\
d &:= y \not\in \{0, 1\} \\
z &:= 1
\end{align*}
\]

Its annotated outcome is disallowed under SRA. Indeed, since T3 reads x = 0 after z = 1, the read of z must read from the write of T2. But then, T4, after reading w = 1 (from T3) cannot read y = 0. However, the semantics described so far allows this outcome as in the following snippet:

\[
\begin{align*}
\{e\} \not\rightarrow \{e\} &\not\rightarrow \{e\} \not\rightarrow \{e\} \\
&\not\rightarrow \{e\} \not\rightarrow \{e\} \not\rightarrow \{e\} \\
&\not\rightarrow \{e\} \not\rightarrow \{e\} \not\rightarrow \{e\} \\
&\not\rightarrow \{e\} \not\rightarrow \{e\} \not\rightarrow \{e\} \\
&\not\rightarrow \{e\} \not\rightarrow \{e\} \not\rightarrow \{e\}
\end{align*}
\]

What went wrong? The problem arises when T3 reads 1 from z. At this point it has two possible futures, o_x^0 · o_y^0 and o_x^0 · o_y^0.
Since read options, consisting of location and value, do not uniquely identify writes, it may read 1 from z, and remain with both \( o_{\tau}' \) and \( o_{\tau} \). Now, it uses one of these options to justify the position of \( o_{\tau}' \) in the list of \( T_{\tau} \), and the other for its own read. However, in a single run of opSRA, when reading 1 from z, \( T_{\tau} \) must pick which write event to read from, and then, either it cannot read \( x = 0 \) or it cannot read \( y = 0 \).

To remedy this problem, we make read options to be more informative. Together with location and value, read options also include the thread identifier that performed the write. When a thread writes, it adds options with its own thread identifier in the different lists. For a thread \( \tau \) to read \( v \) from \( x \), a read option \( o \) with \( val(o) = v \) and \( loc(o) = x \) and some unique writing thread identifier must be the first in every of \( \tau \)’s read-option lists. In this example, the two \( o_{\tau} \) options will have different thread identifiers, which forces \( T_{\tau} \) to discard one of its lists before reading.

Even with thread identifiers, read options do not uniquely identify write events. Nevertheless, as our proof shows, an ambiguity inside the writing thread does not harm the adequacy of the semantics. Roughly speaking, it can be resolved by picking the po-earliest write event, as reading from it enforces the weakest constraints for the rest of the run.

Finally, RMWs behave like an atomic combination of a read and a write, with a slight adaptation of the above semantics. Recall that in opSRA, an RMW may only read from the no-maximal write to the relevant location. To achieve this in loSRA, we include an additional field in read options, which is a binary flag that can be set to either \( R \) or \( RMW \). Intuitively, an RMW value means that the read option is set to read from the no-maximal write. Accordingly, an RMW step may only consume read options marked as \( RMW \). Since write steps to \( x \) replace the no-maximal write to \( x \) in the execution graph, they may choose to mark any of the added read options as \( RMW \), but they can only execute when no read option (of any thread) with location \( x \) is marked as an \( RMW \).

Next, we turn to the formal definitions.

**Notation 5.5 (Sequences).** We use \( \epsilon \) to denote the empty sequence. The length of a sequence \( s \) is denoted by \(|s|\) (in particular \( |\epsilon| = 0 \)). We often identify sequences with their underlying functions (whose domain is \( \{1, \ldots, |s|\} \)), and write \( s(k) \) for the symbol at position \( 1 \leq k \leq |s| \) in \( s \). We write \( \sigma \in s \) if \( \sigma \) appears in \( s \), that is if \( s(k) = \sigma \) for some \( 1 \leq k \leq |s| \). We use "·" for the concatenation of sequences, which is lifted to concatenation of sets of sequences in the obvious way. We identify symbols with sequences of length 1 or their singletons when needed (e.g., in expressions like \( \sigma \cdot S \)).

**Definition 5.6.** Read options, read-option lists and potentials are defined as follows:

1. A read option is a quadruple \( o = (r, x, v, u) \), where \( r \in \text{Tid} \), \( x \in \text{Loc} \), \( v \in \text{Val} \) and \( u \in \{R, RMW\} \). The functions \( \text{tid}, \text{loc}, \text{val} \) and \( \text{rmw} \) return the thread identifier (\( \tau \)), location (\( x \)), value (\( v \)), and RMW flag (\( u \)) of a given read option.
2. A read-option list \( L \) is a sequence of read options.
3. A potential \( B \) is a finite non-empty set of read-option lists.

We define an ordering on read-option lists, which extends to potentials and to assignments of potentials to threads.

**Definition 5.7.** The (overloaded) relation \( \sqsubseteq \) is defined by:

1. on read-option lists: \( L \sqsubseteq L' \) if \( L \) is a (not necessarily contiguous) subsequence of \( L' \);
2. on potentials: \( B \sqsubseteq B' \) if \( \forall L \in B. \exists L' \in B'. L \sqsubseteq L' \) (a.k.a. "Hoare ordering");
3. on functions from Tid to the set of potentials: \( B(\tau) \sqsubseteq B'(\tau) \) for every \( \tau \in \text{Tid} \).

The loSRA memory system is formally defined as follows. Figure 5 illustrates a run of loSRA for the SB program (Ex. 3.5) together with the corresponding run of opSRA.

**Definition 5.8.** loSRA is defined by: \( \text{loSRA}, Q \) is the set of functions \( B \) assigning a potential to every \( \tau \in \text{Tid} \); \( \text{loSRA}, Q_0 = \{\lambda \tau \in \text{Tid}. \{\epsilon\}\} \); and the transitions are given in Fig. 4.

The definition of the write step generally follows the intuitive explanation above. Every read-option list after the write transition is obtained from some previous list, with the addition of \( n \geq 0 \) read options of the current write, provided that: 1. the suffix of the existing list right after the position of the first added option is a read-option list of the writing thread; 2. the lists of the writing thread (which are not discarded in this transition) cannot have options to read from \( x \) besides the ones that are currently added; and 3. the original lists (which are not discarded in this transition) cannot have an \( RMW \) option for \( x \). Note that since the universal quantification is on lists of the new state, the step allows to "duplicate" lists before modifying them, as well as to "discard" complete lists (as often useful when a certain list is needed only as a justification for positioning a read option). We also note that several \( RMW \) options can be added, but only one of them may be later fulfilled, due to condition (3).

**Remark 1.** The write step above insists on having a justification in the form of a complete read-option list of the writing thread \( (L_1 \ldots L_n) \in B(\tau) \). It suffices, however, for the suffix after the first added read option to be a subsequence of some list of the writing thread \( (\{L_1 \ldots L_n\}) \subseteq B(\tau) \). Indeed, this less restrictive step is derivable by combining a lower step and a write step. Note also that for \( \pi = \tau \) (adding read options in the lists of the thread that performed the write), this means that no justification is needed (since \( L_0 \ldots L_n \in B(\tau) \) implies \( \{L_1 \ldots L_n\} \subseteq B(\tau) \)).

The read step requires the first option in all lists in the executing thread’s potential the read to be the same, and consumes it from all these lists. Note that, by definition, the potential \( B'(\tau) \) is non-empty, and so the set \( B(\tau) \) as defined
in the step is non-empty. When all options are consumed, $\tau$ ’s potential consists of a single empty list.

The RMW step is an atomic sequencing of READ and WRITE to the same location. The READ part can only be performed provided that the first option in all lists is marked with RMW.

The LOWER transition allows to remove read options, as well as full read-option lists, at any point. It also allows to add new lists, provided that each new list is “at most as powerful” as some existing list (as used in Remark 1). Intuitively, LOWER can only reduce the possible traces, while it allows us to show that loSRA is a well-structured transition system.

**Example 5.9.** Consider the 2+2W program with its (SRA-disallowed) outcome in Ex. 3.9. To see that this outcome cannot be obtained by loSRA, consider the last write executed in a run of this program. Suppose, w.l.o.g., that it is $y := 2$ by $T_1$. Before executing this write, $T_1$ may not have any read options of location $y$ in its lists. Hence, a read option of the form $(\pi, y, 1, u)$ should be added to $T_1$’s potential after $T_1$ executed $y := 2$. This contradicts our assumption that $y := 2$ was the last executed write.

**Example 5.10.** Consider the 2RMW program with its (SRA-disallowed) outcome in Ex. 3.8. To try to obtain this outcome in loSRA, the $x := 0$ by $T_1$ must add a read option $(T_1, x, 0, RMW)$ in both its own list and in a list of $T_2$. But, the execution of the first RMW, which consumes one of these options, can only proceed after the other option marked with RMW is discarded. Hence, the second RMW cannot execute, and this outcome cannot be obtained by loSRA.

Next, we prove the equivalence of loSRA and opSRA. To do so, we define a relation $\gamma \subseteq loSRA, Q \times opSRA, Q$, formalizing the intuitive simulation relation discussed so far between loSRA’s states and opSRA’s execution graphs. For defining $\gamma$, we first define a “write list” linking the read options in a read-option list $L$ to write events in an execution graph $G$.

**Definition 5.11.** A write list is a sequence $W$ of write events. A write list $W$ is a $(G, L)$-write-list if $|L| = |W|$ and the following hold for every $1 \leq k \leq |W|$ with $L(k) = (r, x, v, u)$:

- $W(k) \in G.W.$
- $\text{tid}(W(k)) = r$, $\text{loc}(W(k)) = x$ and $\text{val}(W(k)) = v$.
- if $u = \text{RMW}$, then $W(k) \notin \text{dom}(G, mo)$.

A write list $W$ is $(G, \tau)$-consistent if, intuitively, the extension of $G$ with a sequence of read events in thread $\tau$ reading from the sequence of write events in $W$ satisfies read-coherence. Thus, we ensure that thread $\tau$ is not already aware of some write that is mo-later than some write of $W$, and that after reading from a write $w_1$ of $W$, thread $\tau$ will not become aware of some write that is mo-later than some write $w_2$ that appears after $w_1$ in $W$. Formally:

**Definition 5.12.** A write list $W$ is called $(G, \tau)$-consistent if $W(k) \notin \text{dom}(G, mo)$; $\tau \in \{\tau, x, y\}$; $\{E' \cup \{W(j) | 1 \leq j < k\}\}$ for every $1 \leq k \leq |W|$.

Now, $\gamma$ relates a loSRA state $B$ with an execution graph $G$ if for each read-option list in $B$ there is an appropriate write list. Formally:
Definition 5.13. A state \( \mathcal{B} \in \text{loSRA,Q} \) matches an execution graph \( G \), denoted by \( \mathcal{B} \triangleleft G \), if for every \( \tau \in \text{Tid} \) and \( L \in \mathcal{B}(\tau) \), there exists a \( (G, \tau) \)-consistent \( (G, L) \)-write-list.

Next, we establish the equivalence of loSRA and opSRA, showing that every trace of opSRA is a trace of loSRA, and vice versa. As loSRA has \( \epsilon \)-transitions (lower), while opSRA does not, the trace equivalence ignores \( \epsilon \)-transitions. Notice that \( \epsilon \)-transitions do not affect reachability of problem states, which only concerns the sequence of labels of the program.

Definition 5.14. Two traces are equivalent if their restrictions to non-\( \epsilon \)-transitions are equal.

Full proofs of Lemmas 5.15 and 5.16 are provided in Appendix C, and are mechanized in the Coq proof assistant.

Lemma 5.15. For every trace of loSRA there is an equivalent trace of opSRA.

Proof Outline. We show that \( \gamma \) constitutes a (weak) forward simulation from loSRA to opSRA. Handling the lower step is easy (new write lists are restrictions of the ones we have). Now, suppose that 1. \( \mathcal{B} \triangleleft G \), witnessed by a \( (G, \pi) \)-consistent \( (G, L) \)-write-list \( W(\pi, L) \) for every \( \pi \in \text{Tid} \) and \( L \in \mathcal{B}(\pi) \); and 2. \( \mathcal{B} \xrightarrow{r,l} \text{loSRA} \mathcal{B}' \). We construct \( G' \) such that \( \mathcal{B}' \triangleright G' \) and \( G \xrightarrow{r,l} \text{opSRA} G' \) (as depicted on the right).

We describe here the write and read steps (the \( \text{raw} \) step is obtained by carefully combining them). For a write step, \( G' \) is trivially constructed by adding a new write event \( w \) to \( G \), placed last in \( \text{po} \) inside the writing thread \( \tau \), and last in \( \text{mo} \) among the writes to the same location. Then, \( G \xrightarrow{r,l} \text{opSRA} G' \) is trivial. To show that \( \mathcal{B}' \triangleright G' \), we construct for every \( \pi \in \text{Tid} \) and \( L' \in \mathcal{B}'(\pi) \), a \( (G', \pi) \)-consistent \( (G', L') \)-write-list \( W' \). The write list \( W' \) maps: 1. the new options—to the new write event \( w \); 2. the "old" options that appear before the first new option—\( \text{as mapped in the existing write list } W \) for the corresponding list in \( \mathcal{B}(\pi) \); and 3. each old option that appears after the first new option—\( \text{to the } \text{mo} \)-maximal write event among a. its mapping in the existing write list \( W \) for the corresponding list in \( \mathcal{B}(\pi) \) and b. its mapping in the existing write list \( W_{\tau} \) for the justifying list in \( \mathcal{B}(\tau) \). Roughly speaking, picking the \( \text{mo} \)-maximal write in the third case ensures that \( W' \) is \( (G', \pi) \)-consistent: In the new state, \( \pi \) might not be able to read from a write that it previously read from (since it "synchronized" with \( \tau \) that may be aware of a later write) and might not be able to read from the a write that \( \tau \) reads from (since \( \pi \) may be already aware of a later write); but, it may read from the later write among these two writes.

In turn, to simulate a \text{read} step of loSRA in opSRA, we need to pick a write event \( w \) from which the added read event \( \tau \) will read from in \( G' \). We pick \( w \) to be the \( \text{po} \)-minimal event among the write events that the \( W(\tau, L) \) lists associate to the first option in some \( L \in \mathcal{B}(\tau) \). (All these options are consumed during the step, and their corresponding writes are all in the same thread as dictated by the thread identifier stored in the read options). The consistency of the \( W(\tau, L) \) lists ensures that \( w \notin \text{dom}(G_{\text{mo}} ; G_{\text{hb}} ; [E]) \), so we can make the \text{read} step in opSRA when reading from \( w \). To show that \( \mathcal{B}' \triangleright G' \), for every thread \( \pi \neq \tau \), we simply rewrite the \text{write list } \( W(\pi, L) \) that we had for \( G \); while for thread \( \tau \) itself, we shift its write lists by one (setting \( W(\tau, L) \triangleq \lambda k. \ W(\tau, L)(1+k) \)), and use the \( \text{po} \)-minimality of \( w \) to establish \( (G', \tau) \)-consistency.

For the converse, we favor \text{backward} simulation, since loSRA requires to "guess" the future, and without knowing the target state, we cannot construct the next step.

Lemma 5.16. For every trace of opSRA there is an equivalent trace of loSRA.

Proof Outline. We show that \( \gamma^{-1} \) constitutes a backward simulation from opSRA to loSRA. For the main proof obligation (depicted on the right), suppose that \( G \xrightarrow{r,l} \text{opSRA} G' \) and \( B' \triangleright G' \), witnessed by a \( (G', \pi) \)-consistent \( (G', L') \)-write-list \( W'(\pi, L') \) for every \( \pi \in \text{Tid} \) and \( L' \in \mathcal{B}'(\pi) \). We construct a state \( \mathcal{B} \) such that \( \mathcal{B} \xrightarrow{r,l} \text{loSRA} \mathcal{B}' \) and \( \mathcal{B} \triangleleft G \).

Again, we describe here the \text{write} and \text{read} steps (the \text{rmw} step is obtained by carefully combining them). First, for a \text{write} step, let \( w \) be the \text{write} event that is added in opSRA’s transition from \( G \) to \( G' \). Roughly speaking, we construct \( \mathcal{B} \) by removing the \text{read} options that were associated to \( w \) in the existing \text{write} lists, and copying the suffix of each \text{read}‐option list after the first \text{read} option associated to \( w \) to the potential of thread \( \tau \). The \text{write} lists for \( \mathcal{B} \) are then induced by those for \( B' \) in the obvious way.

In turn, for a \text{read} step, let \( \tau \) be the \text{read} event that is added in opSRA’s transition from \( G \) to \( G' \), and \( w \) be the \text{write} event that \( \tau \) reads from in \( G' \). Then, we construct \( \mathcal{B} \) by setting \( \mathcal{B} = B'[	au \mapsto \langle \text{tid}(w), \text{loc}(r), \text{val}(r), R \rangle \cdot B'(\tau)] \), and \( \mathcal{B} \xrightarrow{r,l} \text{loSRA} \mathcal{B}' \) follows by definition. Now, to show that \( \mathcal{B} \triangleleft G \), we use the \text{write} lists of \( B' \) for every \( \pi \neq \tau \). For \( \pi = \tau \), we append \( w \) in the beginning of the \text{write} lists of \( B' \).

We conclude with the equivalence of opSRA and loSRA.

Theorem 5.17. For every program \( P \), the set of program states that are reachable under opSRA coincides with the set of program states that are reachable under loSRA.

Proof. Directly follows from Lemmas 5.15 and 5.16.

6 Decidability of the Reachability Problem

We show how loSRA is used for establishing the decidability of the reachability problem under the declarative SRA model (see Def. 3.3). We start with recalling the framework of well-structured transition systems.
Well structured transition systems. A well-quasi-ordering (wqo) on a set $S$ is a reflexive and transitive relation $\preceq$ on $S$ such that for every infinite sequence $s_1, s_2, \ldots$ of elements of $S$, we have $s_i \not\preceq s_j$ for some $i < j$. In a context of a set $S$ and a wqo $\preceq$ on $S$, the upward closure of a set $U \subseteq S$, denoted by $\upsetminus U$, is given by $\{ s \in S \mid \exists u \in U, u \not\preceq s \}$; a set $U \subseteq S$ is called upward closed if $U = \upsetminus U$; and a set $B \subseteq U$ is called a basis of $U$ if $U = \upsetminus B$. The properties of a wqo ensure that every upward closed set has a finite basis.

A well-structured transition system (WSTS) is an LTS $A$ equipped with a wqo $\preceq$ on $A.Q$ that is compatible with $A$, that is: if $q_1 \rightarrow A q_2$ and $q_1 \preceq q_3$, then there exists $q_4 \in A.Q$ such that $q_3 \rightarrow A q_4$ and $q_2 \preceq q_4$. The coverage problem for $(A, \preceq)$ asks whether an input state $q \in A.Q$ is coverable, namely: is some state $q'$ with $q \preceq q'$ reachable in $A$?

Coverability is decidable (see, e.g., [5]) for a WSTS $A$ ordered pairs of $A.Q$’s states is equal, reachability under loSRA is reduced to coverability in $(P_{loSRA}, \subseteq)$, which is decidable by Lemma 6.1 and the framework of [5].

We are now in position to prove our main results.

Corollary 6.3. The SRA reachability problem is decidable.

Proof. Directly follows from Theorems 4.4, 5.17 and 6.2.

Corollary 6.4 (RA race-free reachability). Given a write/write-race-free (see Def. 3.11) program $P$ and a state $p \in P.Q$, it is decidable to check whether $p$ is reachable under RA.

Proof. Directly follows from Thm. 3.12 and Corollary 6.3.

7 Conclusion and Future Work

We established the decidability of reachability under SRA, a fundamental causal consistency memory model. For that matter, we developed a novel operational semantics for SRA and showed that it meets the requirements for decidability of the framework of well-structured transition systems. Besides the theoretical interest, Abdulla et al. [3] demonstrate that similar verification procedures (also of non-primitive recursive complexity) may be actually practical for challenging (even though naturally quite small) algorithms and synchronization mechanisms. We plan to explore this in the future.

Reachability is undecidable under C/C++11’s causal consistency model, RA [2]. Intuitively, this stems from the fact that RA requires to maintain mo separately from the execution order, while SRA allows the execution of writes following hb ∪ mo. (We note that the existing undecidability result crucially employs RMWs, and the decidability of RA without RMW operations is still open.) Since RA and SRA coincide on write/write-race-free programs, and write/write-race freedom can be checked under SRA (Thm. 3.12), our result allows the verification of safety properties under RA for this large and widely used class of programs. Concurrent separation logics [23, 45, 46], designed for verification under RA, are also essentially limited to reason only about write/write-race-free programs and stateless model checking is also significantly simpler with this assumption (see [24, §5]). We also note that it is straightforward to support C/C++11’s non-atomics, with “catch-fire” semantics (i.e., data races are
errors) in addition to release/acquire accesses and sequentially consistent fences (which are modeled as RMWs as in Ex. A.2 in Appendix A). Indeed, as demonstrated in [23], it suffices to check for data races assuming RA semantics. The extension to other fragments of C/C++11, such as relaxed and sequentially consistent accesses, is left to future work.

We believe that the potential-based semantics (both specifically for SRA and as a general idea) may be of independent interest in the development of verification techniques for programs running under weak consistency, including, but not limited to, program logics and model-checking techniques. In particular, we are interested in developing abstraction techniques, as was done for TSO and similar buffer-based models (see, e.g., [25, 43]). Other directions for future work include handling other variants of causally consistent shared-memory (see, e.g., [14]), supporting transactions (to enable, e.g., full verification of client programs under PSI, see §3.1) and studying verification of parametrized programs under causal consistency (which is decidable for TSO [3]).

References


A Additional Examples

Example A.1 (Transitive message passing). po and rf edges equally contribute to hb in causal consistency. Hence, as in Ex. 3.6 (MP), the following outcome is disallowed as well:

\[
\begin{align*}
\text{MP-trans} & \\
x & := 0 & a & := y & \text{ // } & i & b & := x & \text{ // } 1 & c & := x & \text{ // } 0 \times \text{SRA} \\
y & := 1 & x & := 1 & & & & & \end{align*}
\]

Example A.2. RMWs to an otherwise-unused location can be used as fences, as the consistency constraints (of any of the models) imply that hb must totally order G.Wx when, except for one (initialization) write event, all write events to x in G are RMWs. For example, placing such fences forbids the weak outcome of the SB program (Ex. 3.5). An execution graph for this outcome must have the edges as depicted on the right, and any choice of the missing rf-edges (to the two RMW events) will violate a condition of SRA.

\[
\begin{align*}
z & := 0 & x & := 0 & y & := 0 & W(z, 0) & W(x, 0) & W(y, 0) \times \text{SRA} \\
x & := 1 & y & := 1 & a & := \text{FADD}(z, 0) & a & := \text{FADD}(z, 0) & W(x, 1) & \downarrow R(y, 1) \uparrow \text{RF} \downarrow \text{RMW}(z, 0, 0) & R(x, 0) \quad \text{(SBF)} \\
b & := y & \text{ // } 0 & c & := x & \text{ // } 0 & & & & & & & & &
\end{align*}
\]

B Proof of Theorem 3.12

Theorem 3.12. Let P be a program that is write/write-race free under SRA. Then, the sets of states of P that are reachable under SRA and RA coincide.

Proof. Using Prop. 3.10, it suffices to show that every state of P that is reachable under RA is also reachable under SRA.

Let G be the set of all RA-consistent but SRA-inconsistent execution graphs that are generated by P. To show that every state of P that is reachable under RA is also reachable under SRA, it suffices to show that G is empty.

Suppose otherwise and let G be a minimal element in G, in the sense that every proper G.hb-prefix of G is not in G. (A proper G.hb-prefix of G is any execution graph \( \langle E_p, [E_p] ; G.rf ; [E_p] ; G.mo ; [E_p] \rangle \) for some \( E_p \subseteq G.E \) such that \( \text{dom}(G.hb ; [E_p]) \subseteq E_p \).) Note that G cannot be empty, since the empty execution graph G₀ is trivially SRA-consistent.

Let e be a G.hb-maximal event in G.E. Let \( E' = G.E \setminus \{ e \} \), \( rf' = [E'] ; G.rf ; [E'] \) and \( mo' = [E'] ; G.mo ; [E'] \). The minimality of G ensures that \( G' = \langle E', rf', mo' \rangle \) (namely, the restriction of G to E') is SRA-consistent. Hence, our assumption on P ensures that G’ is write/write-race free, thus using irr-hb-mo, it follows that \( mo' \subseteq G'.hb \subseteq G.hb \).

Now, since G is RA-consistent but not SRA-consistent, we have that G does not satisfy irr-hb-mo. Since G’ satisfies irr-hb-mo, it must be the case that there exists w ∈ E such that \( \langle e, w \rangle \in G.mo \) and \( \langle w, e \rangle \in (G.hb \cup mo')^+ \). Since \( mo' \subseteq G.hb \), it follows that \( \langle e, e \rangle \in G.mo \cup G.hb \). Hence, G does not satisfy write-coherence. This contradicts the fact that G is RA-consistent.

C Equivalence of IoSRA and opSRA – Full Proofs

The following alternative formulation of the write step will be convenient to use in our proofs. This formulation “works backwards”—choosing read options to omit from the target state for reaching the source state. Each such possibility is an “index choice”:

Definition C.1. An index choice for a state \( B' \in \text{IoSRA,Q} \) is a function \( \mathcal{P} \) assigning a set \( \mathbb{P}(\pi, L') \subseteq \{ 1, \ldots, |L'| \} \) to every \( \pi \in \text{Tid} \) and \( L' \in B'(\pi) \). An index choice \( \mathcal{P} \) for \( B' \) justifies a \( \langle \tau, W(x, \omega_0) \rangle \)-step, denoted by \( \mathcal{P} \models \langle \tau, W(x, \omega_0) \rangle \), if the following hold for every \( \pi \in \text{Tid} \) and \( L' \in B'(\pi) \):

- For every \( k \in \mathbb{P}(\pi, L') \), we have \( L'(k) = \langle \tau, x, \omega_0, R \rangle \) or \( L'(k) = \langle \tau, x, \omega_0, \text{RMW} \rangle \).
For every \( k \in \{1, \ldots, |L'|\} \setminus P(\pi, L') \):
- If \( \text{loc}(L'(k)) = x \), then \( \text{rmw}(L'(k)) = \emptyset \).
- If \( k > \min P(\pi, L') \) or \( \pi = \tau \), then \( \text{loc}(L'(k)) \neq x \).

Now, a predecessor of \( B' \) with respect to a write step intuitively satisfies two constraints: 1. For each list \( L' \) of \( B' \), there is a corresponding list \( L \) in \( B \) that possibly lack some read options of the form \( (\pi, x, v_\eta, u) \), corresponding to the new read options of \( B' \); and 2. If a list \( L' \) is different from the corresponding list \( L \) then there is a list of \( \tau \) in \( B \) that justifies this difference. Notice that \( B \) may have arbitrary additional lists in addition to the above mandatory lists.

**Notation C.2** (List operations). For a list \( L \) and a set \( P \subseteq \{1, \ldots, |L|\} \) of positions in \( L \), we define:
- \( L \setminus P \) is the list derived from \( L \) by removing from it the positions in \( P \). The mapping of the positions of \( L \) that are not in \( P \) to their matching positions in \( L \setminus P \) is denoted by \( \text{Map}_{(L,P)} \) (formally, \( \text{Map}_{(L,P)}(k) \triangleq \lambda k \in \{1, \ldots, |L|\} \setminus P. k - |\{j \in P \mid j < k\}| \)).
- \( L \setminus P \) further removes from \( L \) the positions before the first position in \( P \), namely returns the list \( L \setminus (P \cup \{1, \ldots, \min(P) - 1\}) \) (undefined if \( P = \emptyset \)). The mapping of the positions of \( L \) that are not in \( P \) and not before the first position in \( P \) to their matching positions in \( L \setminus P \) is denoted by \( \text{MMap}_{(L,P)} \) (formally, \( \text{MMap}_{(L,P)}(k) = \min(P) - \min(P) + 1 \)).

**Definition C.3.** The source of \( B' \) w.r.t. a thread \( \tau \) and an index choice \( \mathcal{P} \) for \( B' \), denoted by \( \text{src}(B', \tau, \mathcal{P}) \), is given by:

\[
\text{src}(B', \tau, \mathcal{P}) \triangleq \lambda \pi \in \text{Tid}.
\begin{align*}
\{L' \setminus \mathcal{P}(\pi, L') \mid L' \in B'(\pi)\} & \quad \pi \neq \tau \\
\{L' \setminus \mathcal{P}(\tau, L') \mid L' \in B'(\tau)\} & \quad \pi = \tau \\
\{L' \setminus \mathcal{P}(\eta, L') \mid \mathcal{P}(\eta, L') \neq \emptyset, \eta \in \text{Tid} \text{ and } L' \in B'(\eta)\} &
\end{align*}
\]

The following proposition follows directly from our definitions.

**Proposition C.4.** \( B \xrightarrow{\tau,W(x,v_\eta)}_{\text{loSRA}} B' \) iff there exists an index choice \( \mathcal{P} \) for \( B' \) such that \( \mathcal{P} \models (\tau, W(x, v_\eta)) \) and \( \text{src}(B', \tau, \mathcal{P}) \subseteq B(\pi) \) for every \( \pi \in \text{Tid} \).

**Lemma 5.15.** For every trace of \( \text{loSRA} \) there is an equivalent trace of \( \text{opSRA} \).

**Proof.** We show that \( \gamma \) constitutes a forward simulation relation from \( \text{loSRA} \) to \( \text{opSRA} \). First, the initial states clearly match: we have \( \lambda \tau. \{e\} \gamma G_0 \). Now, suppose that \( B \gamma G \) and \( B \xrightarrow{\tau}_{\text{loSRA}} B' \). We show that there exists \( G' \) such that \( B' \gamma G' \) and \( G \xrightarrow{\tau}_{\text{opSRA}} G' \). Consider the possible cases:

- \( I = W(x,v_\eta) \): Let \( w = \text{NextEvent}(G.E, I, \tau) \). Let \( G' \) be the execution graph induced by \( G.E = G.E \cup \{w\} \), \( G'.r = G.r \) and \( G'.m = G.m \cup (G.W_x \times \{w\}) \). By definition, we have \( G \xrightarrow{\tau}_{\text{opSRA}} G' \). We show that \( B' \gamma G' \). By Prop. C.4, since \( B \xrightarrow{\tau}_{\text{loSRA}} B' \), there exists an index choice \( \mathcal{P} \) for \( B' \) that justifies a \( (\tau, l) \)-step, such that \( \text{src}(B', \tau, \mathcal{P})(\pi) \subseteq B(\pi) \) for every \( \pi \in \text{Tid} \). Let \( \pi \in \text{Tid} \) and \( L' \in B'(\pi) \). We construct a \( \langle G', \pi' \rangle \)-consistent \( \langle G', L' \rangle \)-write-list \( W' \). Let \( P \triangleq \mathcal{P}(\pi', L'), L \triangleq L' \setminus P, f \triangleq \text{Map}_{(L',P)}, L_\tau \triangleq \{L' \setminus P \} \) and \( f_\tau \triangleq \text{MMap}_{(L',P)} \) (the last two are only defined if \( P \neq \emptyset \)). Since \( B \gamma G \), there exist a \( \langle G, L \rangle \)-consistent \( \langle G, L \rangle \)-write-list \( W \), and a \( \langle G, \tau \rangle \)-consistent \( \langle G, L_\tau \rangle \)-write-list \( W_\tau \). We define \( W' \) as follows:

\[
W' \triangleq \lambda k \in \{1, \ldots, |L'|\}.
\begin{align*}
\text{w} & \quad k \in P \\
W(f(k)) & \quad k < \min(P) \\
\max_{G.m} \{W(f(k)), W_\tau(f_\tau(k))\} & \text{otherwise}
\end{align*}
\]

It is easy to see that \( W' \) is a \( \langle G', L' \rangle \)-write-list. In particular, to show that \( \text{rmw}(L'(k)) = \text{rmw} \) implies \( W'(k) \notin \text{dom}(G'.m) \), we use the fact that \( \mathcal{P} \) justifies a \( (\tau, l) \)-step, and so for every \( k \in \{1, \ldots, |L'|\} \setminus P \), we have that \( \text{rmw}(L'(k)) = \text{rmw} \) implies \( \text{loc}(L'(k)) \neq x \).
We show that $W'$ is $(G', \pi)$-consistent.
Let $1 \leq k \leq |L'|$. We prove that $W'(k) \notin \text{dom}(G'.\text{mo} ; G'.\text{hb}^2 ; [E^\pi \cup \{ W'(j) \mid 1 \leq j < k \}])$. Suppose otherwise. First, note that we cannot have $k = \ell$, since $w$ is a maximal element in $G'.\text{mo}$. Let $w_{\pi} = W(f(k))$ and $w_{\tau} = W_r(f_r(k))$ (the latter is only defined if $k > \min(P)$). Consider the two possible cases:

$\triangleright$ $W'(k) \in \text{dom}(G'.\text{mo} ; G'.\text{hb}^2 ; [E^\pi])$: The definition of $W'$ ensures that $\langle w_{\pi}, W'(k) \rangle \in G'.\text{mo}^2$, and so $w_{\pi} \in \text{dom}(G'.\text{mo} ; G'.\text{hb}^2 ; [E^\pi])$. Since $W$ is $(G, \pi)$-consistent, we have $w_{\pi} \notin \text{dom}(G.\text{mo} ; G.\text{hb}^2 ; [E^\pi])$, and therefore it must be the case that $\pi = \tau$ and $\langle w_{\pi}, w \rangle \in G'.\text{mo}$. Hence, $\text{loc}(w_{\pi}) = x$, and so $\text{loc}(L'(k)) = x$, which contradicts the fact that $P$ justifies a $(\tau, l)$-step.

$\triangleright$ $\langle W'(k), W'(j) \rangle \in G'.\text{mo} ; G'.\text{hb}^2$ for some $1 \leq j < k$. Consider the two possible cases:

$\triangleright$ $W'(j) = w$: In this case we must have $k > \min(P)$, and so $W'(k) = \text{max}_{G.\text{mo}} \{ w_{\pi}, w_{\tau} \}$. Hence, we have $\langle w_{\tau}, W'(k) \rangle \in G.\text{mo}^2$, and so $\langle w_{\tau}, w \rangle \in G'.\text{mo} ; G'.\text{hb}^2$. Now, if $\langle w_{\tau}, w \rangle \in G'.\text{mo} ; G'.\text{hb}^2$, then we also have $w_{\tau} \in \text{dom}(G.\text{mo} ; G.\text{hb}^2 ; [E^\pi])$, which contradicts the fact that $W_r$ is $(G, \tau)$-consistent. Therefore, we have $\langle w_{\tau}, w \rangle \in G'.\text{mo}$. Hence, $\text{loc}(w_{\pi}) = x$, and so $\text{loc}(L'(k)) = x$, which contradicts the fact that $P$ justifies a $(\tau, l)$-step.

$\triangleright$ $W'(j) \neq w$: In this case we must have $\langle W'(k), W'(j) \rangle \in G.\text{mo} ; G.\text{hb}^2$. The definition of $W'$ ensures that $\langle w_{\pi}, W'(k) \rangle \in G.\text{mo}^2$, and so $\langle w_{\pi}, w \rangle \in G'.\text{mo} ; G'.\text{hb}^2$. Now, since $W$ is $(G, \pi)$-consistent, we cannot have $W'(j) = W(f(j))$. Hence, $j > \min(P)$ and $W'(j) = W_r(f_r(j))$. Let $w_{\pi}' = W_r(f_r(j))$. It follows that $k > \min(P)$, and so $\langle w_{\pi}', W'(k) \rangle \in G.\text{mo}^2$. Hence, we have $\langle w_{\pi}', w \rangle \in G.\text{mo} ; G.\text{hb}^2$. This contradicts the fact that $W_r$ is $(G, \tau)$-consistent.

• $l = R(x, v_0)$: By definition, since $B \xrightarrow{\text{opSRA}^*} B'$, there exists a read option $o$ with $\text{loc}(o) = x$ and val$(o) = v_0$ such that $B(\tau) = o \cdot B'(\tau)$. Since $B \cup G$, for every $L \in B(\tau)$ there exists a $(G, \tau)$-consistent $(G, L)$-write-list $W_L$. Let $A = \{ W_L(1) \mid L \in B(\tau) \}$. Since $B(\tau)$ is non-empty, we know that $A$ is not empty. Since each $W_L$ is a $(G, L)$-write-list, we have that $\text{tid}(w) = \text{tid}(o)$ for every $w \in A$. Hence, $G.\text{po}$ totally orders $A$. Let $w = \text{min}_{G.\text{po}} A$ and let $L_m = B(\tau)$ such that $w = W_{L_m}(1)$. Let $r = \text{NextEvent}(G.\text{E}, \epsilon, \tau, l)$ and let $G'$ be the execution graph obtained by $G'.\text{E} = G.\text{E} \cup \{ r \}, G'.\text{rf} = G.\text{rf} \cup \{ \langle w, r \rangle \}$ and $G'.\text{mo} = G.\text{mo}$. We show that $G \xrightarrow{\text{opSRA}^*} G'$. By definition, it suffices to show the following:

$\triangleright$ $w \in G.\text{Wx}$ and val$_L(w) = v_0$: We have $w = W_{L_m}(1)$, and since $W_{L_m}$ is a $(G, L_m)$-write-list, we have that $w \in G.\text{Wx}$, $\text{loc}(o) = \text{loc}(W_{L_m}(1)) = \text{loc}(L_m(1)) = \text{loc}(o) = x$ and val$_L(w) = \text{val}(W_{L_m}(1)) = \text{val}(L_m(1)) = \text{val}(o) = v_0$.

$\triangleright$ $w \notin \text{dom}(G.\text{mo} ; G.\text{hb}^2 ; [E^\pi])$: Since $W_{L_m}$ is $(G, \tau)$-consistent and $w = W_{L_m}(1)$, we cannot have $w \in \text{dom}(G.\text{mo} ; G.\text{hb}^2 ; [E^\pi])$.

It remains to show that $B' \cup G'$. Let $\pi \in \text{Tid}$ and $L' \in B'(\pi)$. We define a $(G', \pi)$-consistent $(G', L')$-write-list. Consider two cases:

$\triangleright$ $\pi \neq \tau$: By definition, since $B \xrightarrow{\text{opSRA}^*} B'$, we have $L' \in B(\pi)$. Since $B \cup G$, there exists a $(G, \pi)$-consistent $(G, L')$-write-list $W$. It is easy to see that $W$ is a $(G', L')$-write-list. We show that $W$ is also $(G', \pi)$-consistent. Let $1 \leq k \leq |L'|$. Suppose by contradiction that $W(k) \in \text{dom}(G'.\text{mo} ; G'.\text{hb}^2 ; [E^\pi \cup \{ W(j) \mid 1 \leq j < k \}])$. It follows that

$$W(k) \in \text{dom}(G.\text{mo} ; G.\text{hb}^2 ; [E^\pi \cup \{ W(j) \mid 1 \leq j < k \}]).$$

This contradicts the fact that $W$ is $(G, \pi)$-consistent.

$\triangleright$ $\pi = \tau$: Let $L = o \cdot L'$. Then, $L \in B(\tau)$. Let $W' = \lambda k \in \{ 1, ..., |L'| \}$. $W_L(1 + k)$. It is easy to see that $W'$ is a $(G', L')$-write-list. We show that $W'$ is $(G', \tau)$-consistent. Suppose by contradiction that $W'(k) \in \text{dom}(G'.\text{mo} ; G'.\text{hb}^2 ; [E^\pi \cup \{ W'(j) \mid 1 \leq j < k \}])$.

Now, if $W'(k) \in \text{dom}(G'.\text{mo} ; G'.\text{hb}^2 ; [E^\pi \cup \{ W'(j) \mid 1 \leq j < k \}])$, it follows that

$$W_L(1 + k) \in \text{dom}(G.\text{mo} ; G.\text{hb}^2 ; [E^\pi \cup \{ W_L(1 + j) \mid 1 \leq j < k \}]),$$

which contradicts the fact that $W_L$ is $(G, \tau)$-consistent. Hence, we must have $\langle W'(k), w \rangle \in G.\text{mo} ; G.\text{hb}^2$. Since $L(1) = o$, the definition of $w$ ensures that $\langle w, W_L(1) \rangle \in G.\text{po}^2$. It follows that $\langle W_L(1 + k), W_L(1) \rangle \in G.\text{mo} ; G.\text{hb}^2$, which again contradicts the fact that $W_L$ is $(G, \tau)$-consistent.
\( l \triangleq \text{RMW}(x, v_R, v_w) \)

By definition, since \( \mathcal{B} \xrightarrow{r_I} \text{loSRA} \mathcal{B}' \), there exists a read option \( o \) with \( \text{loc}(o) = x \), \( \text{val}(o) = v_R \) and \( \text{rmw}(o) = \text{RMW} \) such that \( L(1) = o \) for every \( L \in \mathcal{B}(\tau) \). Since \( \mathcal{B} \triangleright G \), for every \( L \in \mathcal{B}(\tau) \) there exists a \( \langle G, \tau \rangle \)-consistent \( (G, L) \)-write-list \( W_L \). Moreover, since \( \text{rmw}(o) = \text{RMW} \), we have \( W_L(1) = \max_{G,\text{mo}} G.W_x \) for every \( L \in \mathcal{B}(\tau) \).

Let \( w = \max_{G,\text{mo}} G.W_x.e = \text{NextEvent}(G.E, \tau, l) \) and \( G' \) be the execution graph given by \( G'.E = G.E \cup \{ e \} \), \( G'.r_f = G.r_f \cup \{ (w, e) \} \) and \( G'.\text{mo} = G.\text{mo} \cup (G.W_x \times \{ e \}) \).

For showing that \( \mathcal{B} \xrightarrow{r_I} \text{loSRA} \mathcal{B}' \), it suffices, by definition, to show that \( \text{val}_w(w) = v_R \). Indeed, since \( \mathcal{B}(\tau) \) is (by definition) non-empty, we can take some \( L \in \mathcal{B}(\tau) \). We have \( w = W_L(1) \), and since \( W_L \) is a \( \langle G, L \rangle \)-write-list, we have that \( \text{val}_w(w) = \text{val}_w(W_L(1)) = \text{val}(1) = v_R \).

It remains to show that \( \mathcal{B}' \triangleright G' \). Using Prop. C.4, since \( \mathcal{B} \xrightarrow{r_I} \text{loSRA} \mathcal{B}' \), we know that there exists an index choice \( \mathcal{P} \) for \( \mathcal{B}' \) that justifies a \( \langle \tau, W(x, w) \rangle \)-step, such that \( \text{src}(\mathcal{B}', \tau, \mathcal{P}) \) \( (\pi) \subseteq \mathcal{B}(\tau) \) for every \( \pi \in \text{Tid} \setminus \{ \tau \} \) and \( o \cdot \text{src}(\mathcal{B}', \tau, \mathcal{P})(\pi) \subseteq \mathcal{B}(\tau) \).

Let \( \pi \in \text{Tid} \) and \( L' \in \mathcal{B}'(\tau) \). We construct a \( \langle G', \pi \rangle \)-consistent \( \langle G', L' \rangle \)-write-list \( W' \). Let \( P \triangleq \mathcal{P}(\pi, L') \) and (the last two are only defined if \( P \neq \emptyset \)):

\[
\begin{align*}
L & \triangleq L' \setminus P & \pi \neq \tau \\
o \cdot (L' \setminus P) & \pi = \tau \\
L_\pi & \triangleq o \cdot L' \parallel P & f \triangleq \begin{cases} \\
\text{Map}(L', P) & \pi \neq \tau \\
\lambda k \in \{1, \ldots, |L'| \} \setminus P. \text{Map}(L', P)(k) + 1 & \pi = \tau \\
\end{cases} \\
f' \triangleq \lambda k \in \{\min(P), \ldots, |L'| \} \setminus P. \text{Map}(L', P)(k) + 1 \\
\end{align*}
\]

Since \( \mathcal{B} \triangleright G \), there exist a \( \langle G, \pi \rangle \)-consistent \( \langle G, L_\pi \rangle \)-write-list \( W_\pi \), and a \( \langle G, \tau \rangle \)-consistent \( \langle G, L_\pi \rangle \)-write-list \( W_\tau \). We define \( W' \) as follows:

\[
W' \triangleq \begin{cases} \\
\{ e \} & k \in P \\
W(f(k)) & k < \min(P) \\
\max_{G,\text{mo}} \{ W(f(k)), W_\tau(f_\tau(k)) \} & \text{otherwise} \\
\end{cases}
\]

It is easy to see that \( W' \) is a \( \langle G', L' \rangle \)-write-list. In particular, to show that \( \text{rmw}(L'(k)) = \text{RMW} \) implies \( \text{rmw}(L'(k)) \notin \text{dom}(G'.\text{mo}), [E^\pi \cup \{ W'(j) \mid 1 \leq j < k \}] \), suppose otherwise. First, note that we cannot have \( k \in P \), since \( e \) is a maximal element in \( G'.\text{mo} \). Let \( w_\pi = W(f(k)) \) and \( w_\tau = W_\tau(f_\tau(k)) \) (the latter is only defined if \( k > \min(P) \)).

Consider the two possible cases:

\(- W'(k) \in \text{dom}(G'.\text{mo} ; G'.R \cdot [E^\pi]) \):

The definition of \( W' \) ensures that \( \langle w_\pi, W'(k) \rangle \in G'.\text{mo} \), and so \( w_\pi \in \text{dom}(G'.\text{mo} ; G'.R \cdot [E^\pi]) \). Since \( W \) is \( \langle G, \pi \rangle \)-consistent, we have \( w_\pi \notin \text{dom}(G.\text{mo} ; G.\text{hb}) ; [E^\pi] \), and therefore it must be the case that \( \langle w_\pi, e \rangle \in G'.\text{mo} ; (G.\text{hb} ; G'.R \cdot [E^\pi]) \) and \( \pi = \tau \). Since \( \mathcal{P} \) justifies a \( \langle \tau, W(x, w) \rangle \)-step, we have that \( \text{loc}(L'(k)) \neq x \) and \( \text{loc}(w_\pi) \neq x \). Hence, \( \langle w_\pi, e \rangle \notin G'.\text{mo} \), and so we have \( \langle w_\pi, e \rangle \in G.\text{mo} ; G.\text{hb} ; G'.R \cdot [E^\pi] \), namely \( \langle w_\pi, w \rangle \in G.\text{mo} ; G.\text{hb} \). However, \( W(1) = w \), contradicting the \( \langle G, \pi \rangle \)-consistency of \( W' \).

\(- \langle W'(k), W'(j) \rangle \in G'.\text{mo} ; G'.R \cdot [E^\pi] \) for some \( 1 \leq j < k \). Consider the two possible cases:

\(* W'(j) = e \):

In this case we must have \( k > \min(P) \), and so \( W'(k) = \max_{G,\text{mo}} \{ w_\pi, w_\tau \} \). There are three possibilities:

\(- W'(k) = w \):

Then \( \text{loc}(w_\tau) = \text{loc}(L'(k)) = x \), which contradicts the fact that \( \mathcal{P} \) justifies a \( \langle \tau, W(x, w) \rangle \)-step.

\(- \langle W'(k), w \rangle \in G'.\text{mo} ; G'.R \cdot [E^\pi] \):

This contradicts the \( \langle G, \pi \rangle \)-consistency of \( W_\tau \), as \( W_\tau(1) = w \) and \( \langle w_\pi, W'(k) \rangle \in G'.\text{mo} \), implying that \( \langle w_\pi, W_\tau(1) \rangle \in G.\text{mo} ; G.\text{hb} \).

\(- \langle W'(k), e \rangle \in G'.\text{mo} ; G'.R \cdot [E^\pi] \):

This also contradicts the \( \langle G, \pi \rangle \)-consistency of \( W_\tau \), as we get that \( w_\tau \notin \text{dom}(G.\text{mo} ; G.\text{hb}) ; [E^\pi] \).
Recall that a backward simulation from an LTS $B$ is a relation $\gamma \subseteq Q \times Q$ such that $1. \gamma^{-1}$ is total (for every $q \in Q$ there exists $\gamma^{-1}(q) \in Q$).

The two first requirements of a backward simulation clearly hold for $\gamma$. 2. Consider a state $B \cup \gamma G_\emptyset Q_\emptyset$. By the definition of $\gamma$, it should be possible to link every read option of $B$ to some write event of $G_\emptyset$. Since there are no write events in $G_\emptyset$, there cannot be read options in $B$, having $B \cup \gamma G Q_\emptyset$.

We move to the third requirement. Suppose that $B \cup \gamma G Q'$ such that $B \cup \gamma G$. We construct a state $B$ such that $B \cup \gamma SRA B'$. Consider the possible cases:

- $l = W(x, o_w)$:
  
  Let $w = \text{NextEvent}(G.E, \tau, l)$. Since $B \cup \gamma SRA G'$, we have $G'.E = G.E \cup \{w\}$, $G'.rf = G.rf$ and $G'.mo = G.mo \cup (G.W_x \times \{w\})$. Since $B \cup \gamma G'$, for every $x \in \text{Tid}$ and $l' \in B'(\pi)$ there exists a $(G', \pi)$-consistent $(G', L')$-write-list $W_{(\pi, L')}$. Let $P$ be the index choice for $B'$ that assigns the set of “new” positions in $B'$:

$$P \triangleq \lambda \pi \in \text{Tid}, L' \in B'(\pi). \{1 \leq k \leq |L'| \mid W_{(\pi, L')}'(k) = w\}.$$

Then, we define $B \cup \gamma SRA B'$, it suffices to prove that $P$ justifies a $(\tau, W(x, o_w))$-step. Thus, we show that the following hold for every $x \in \text{Tid}$ and $l' \in B'(\pi)$, where $P = P(\tau, L')$ and $W' = W_{(\pi, L')}):(1)$

- Let $k \in P$. To see that $L'(k) \in \{(x, x, o_w, R), (x, x, o_w, \text{RMW})\}$, note that since $k \in P$, we have $W(k) = w$, and since $W$ is a $(G', L')$-write-list, we must have $\tau = \text{tid}(w) = \text{tid}(L'(k))$, $x = \text{loc}(L'(k)) = x$ and $o_w = \text{val}_x(w) = \text{val}_x(L'(k))$.

- Let $k \in \{1, \ldots, |L'|\} \setminus P$, such that $\text{loc}(L'(k)) = x$. We show that $\text{rmw}(L'(k)) = R$. Let $w = W'(k)$. Since $k \notin P$, we have $w \neq w$. Since $G'.mo = G.mo \cup (G.W_x \times \{w\})$, it follows that $(w', w) \in G'.mo$. However, since $W$ is a $(G', L')$-write-list, if $\text{rmw}(L'(k)) = \text{RMW}$, then we must have $w' = \text{max}_{G'.mo} G'.W_x$, reaching a contradiction.

- Let $k \in \{m + 1, \ldots, |L'|\} \setminus P$ where $m = \text{min}(P)$, we show that $\text{loc}(L'(k)) \neq x$. Suppose otherwise. Let $w' = W'(k)$. Since $k \notin P$, we have $w' \neq w$. Hence, since $G'.mo = G.mo \cup (G.W_x \times \{w\})$, we have $(w', w) \in G'.mo$. Thus, $(w', W'(m)) \in G'.mo; G'.hb^2$. Since $k > m$, this contradicts the fact that $W'$ is $(G', \pi)$-consistent.

- Suppose that $\pi = \tau$ and let $k \in \{1, \ldots, |L'|\} \setminus P$. We show that $\text{loc}(L'(k)) \neq x$. Suppose otherwise. Let $w' = W'(k)$. Since $k \notin P$, we have $w' \neq w$. Hence, since $G'.mo = G.mo \cup (G.W_x \times \{w\})$, we have $(w', w) \in G'.mo$. Thus, we have $w' \in \text{dom}(G'.mo; G'.hb^2; [E^*])$, which contradicts the fact that $W'$ is $(G', \pi)$-consistent.

Lemma 5.16. For every trace of opSRA there is an equivalent trace of loSRA.

Proof: We show that $\gamma^{-1}$ constitutes a backward simulation from opSRA to loSRA.3

3 Recall that a backward simulation from an LTS $B$ to an LTS $A$ is a relation $R \subseteq A \times B \times Q_\emptyset$ such that $1. R$ is total (for every $q \in A.Q_\emptyset$ we have $\langle q, p \rangle \in R$ for some $p \in B.Q_\emptyset$), 2. if $(q, p) \in R$ and $q \in A.Q_\emptyset$, then $p \in B.Q_\emptyset$, and 3. if $q \sim A q'$ and $(q', p') \in R$, then there exists $p \in B.Q_\emptyset$ such that $\langle q, p \rangle \sim_A B p'$ and $(q, p) \in R$. 

* $W'(j) \neq e$: In this case, we must have $\langle W'(k), W'(j) \rangle \in G.mo; G.hb^2$. The definition of $W'$ ensures that $(w_\pi, W'(k)) \in G.mo^2$, and so $(w_\pi, W'(j)) \in G.mo; G.hb^2$. Now, since $W$ is $(G, \pi)$-consistent, we cannot have $W'(j) = W(f(j))$. Let $w'_\pi = W_\pi(f(j))$. Hence, $j > \text{min}(P)$ and $W'(j) = w'_\pi$. It follows that $k > \text{min}(P)$, and so $(w_\pi, W'(k)) \in G.mo^2$. Hence, we have $(w_\pi, w'_\pi) \in G.mo; G.hb^2$. This contradicts the fact that $W_\pi$ is $(G, \pi)$-consistent.
It remains to show that $B \supseteq G$. Let $\pi \in \text{Tid}$ and $L \in B(\pi)$. We show that there exists a $(G, \pi)$-consistent $(G, L)$-write-list $W$. Following the construction of $B$, one of the following holds:
- $L = L' \setminus \mathcal{P}(\pi, L')$ for some $L' \in B'(\pi)$. Let $P = \mathcal{P}(\pi, L')$, $W' = W'_{(\pi, L')}$ and $f = \text{Map}^{-1}_{(\pi, L')}$. We define $W = \lambda k \in \{1, \ldots, |L|\}$. $W'((f(k))$. Using the fact that $W'$ is a $(G', L')$-write-list, it is easy to see that $W$ is a $(G, L)$-write-list. (In particular, note that $\text{rmw}(L(k)) \neq \text{RMW}$ whenever $1 \text{oc}(L(k)) = x$.)

It remains to show that $W$ is $(G, \pi)$-consistent, namely to prove that for every $k$, we have $W(k) \notin \text{dom}(G, \pi), G, L$. By definition, $\mathcal{R}(\pi, L')$ for some $\pi \in \text{Tid}$ and $L' \in B'(\eta)$. Let $P = \mathcal{P}(\eta, L')$, $m = \text{min}(P)$, $W' = W'_{(\eta, L')}$. We define $W = \lambda k \in \{1, \ldots, |L|\}$. $W'((f(k))$. Using the fact that $W'$ is a $(G', L')$-write-list, it is easy to see that $W$ is a $(G, L)$-write-list. (In particular, note that $\text{rmw}(L(k)) \neq \text{RMW}$ whenever $1 \text{oc}(L(k)) = x$.)

It remains to show that $W$ is $(G, \pi)$-consistent, namely to prove that for every $k$, we have $W(k) \notin \text{dom}(G, \pi), G, L$. By definition, $\mathcal{R}(\pi, L')$ for some $\pi \in \text{Tid}$ and $L' \in B'(\eta)$. Let $P = \mathcal{P}(\eta, L')$, $m = \text{min}(P)$, $W' = W'_{(\eta, L')}$. We define $W = \lambda k \in \{1, \ldots, |L|\}$. $W'((f(k))$. Using the fact that $W'$ is a $(G', L')$-write-list, it is easy to see that $W$ is a $(G, L)$-write-list. (In particular, note that $\text{rmw}(L(k)) \neq \text{RMW}$ whenever $1 \text{oc}(L(k)) = x$.)

It remains to show that $W$ is $(G, \pi)$-consistent, namely to prove that for every $k$, we have $W(k) \notin \text{dom}(G, \pi), G, L$. By definition, $\mathcal{R}(\pi, L')$ for some $\pi \in \text{Tid}$ and $L' \in B'(\eta)$. Let $P = \mathcal{P}(\eta, L')$, $m = \text{min}(P)$, $W' = W'_{(\eta, L')}$. We define $W = \lambda k \in \{1, \ldots, |L|\}$. $W'((f(k))$. Using the fact that $W'$ is a $(G', L')$-write-list, it is easy to see that $W$ is a $(G, L)$-write-list. (In particular, note that $\text{rmw}(L(k)) \neq \text{RMW}$ whenever $1 \text{oc}(L(k)) = x$.)
Let $\mathcal{P}$ be the index choice for $\mathcal{B}'$ that assigns the set of “new” positions in $\mathcal{B}'$:

$$\mathcal{P} \triangleq \lambda \pi \in \text{Tid}, L' \in \mathcal{B}'(\pi). \{1 \leq k \leq |L'| \mid W'_{(\pi,L')}(k) = e\}.$$ 

Then, we define:

$$\mathcal{B} \triangleq \lambda \pi \in \text{Tid}. \begin{cases} o \cdot \text{src}(\mathcal{B}', \tau, \mathcal{P})(\tau) & \pi = \tau \\ \text{src}(\mathcal{B}', \tau, \mathcal{P})(\pi) & \pi \neq \tau \end{cases}$$

where $o$ is the read option given by $o \triangleq (\text{tid}(w), x, v, \text{RMW})$.

The arguments for why $\mathcal{B} \xrightarrow{\text{RMW}} \mathcal{B}'$ are analogous to those of the write case. Using Prop. C.4, to show that $\mathcal{B} \xrightarrow{\text{RMW}} \mathcal{B}'$, it suffices to prove that $\mathcal{P}$ justifies a $\langle \tau, W(x, v) \rangle$-step. This is done exactly as in the write case.

It remains to show that $\mathcal{B} \supseteq \mathcal{G}$. Let $\pi \in \text{Tid}$ and $L \in \mathcal{B}(\pi)$. We show that there exists a $\langle G, \pi \rangle$-consistent $\langle G, L \rangle$-write-list $W$. Following the construction of $\mathcal{B}$, one of the following holds:

- $L = L' \setminus \mathcal{P}(\pi, L')$ for some $L' \in \mathcal{B}'(\pi)$. This case is exactly the same as the analogous case in the write step.
- $\pi = \tau$ and $L = o \cdot (L' \setminus \mathcal{P}(\tau, L'))$ for some $L' \in \mathcal{B}'(\tau)$. Let $P = \mathcal{P}(\tau, L')$, $W' = W'_{(\tau, L')}$ and $f = \lambda k \in \{2, \ldots, |L|\}$. Map$^{-1}$($P_{(L', P)}$)($k - 1$). We define

$$W \triangleq \lambda k \in \{1, \ldots, |L|\}. \begin{cases} w & k = 1 \\ W'(f(k)) & k > 1 \end{cases}$$

Using the fact that $W'$ is a $\langle G', L' \rangle$-write-list and that $w = \max_{\text{max}_{\text{Go}}} W_x$, it is easy to see that $W$ is a $\langle G, L \rangle$-write-list. (In particular, note that for $k > 1$, $\text{rmw}(L(k)) \neq \text{RMW}$ whenever $\text{lcc}(L(k)) = x$.) It remains to show that $W$ is $(G, \tau)$-consistent, namely to prove that for every $k \in \{1, \ldots, |L|\}$, we have $W(k) \notin \text{dom}(G, \text{mo} ; G, \text{hb}^\ominus ; [E^\tau \cup \{W(j) \mid 1 \leq j < k\}])$. For $k = 1$, this is trivial since $W(1) = w = \max_{\text{max}_{\text{Go}}} W_x$. Let $k \in \{2, \ldots, |L|\}$. If $W(k) \in \text{dom}(G, \text{mo} ; G, \text{hb}^\ominus ; [E^\tau])$ then since $G, \text{mo} \subseteq G', \text{mo}$ and $G, \text{hb} \subseteq G', \text{hb}$, we have $W'(f(k)) \in \text{dom}(G', \text{mo} ; G', \text{hb}^\ominus ; [E^\tau])$, which contradicts the $\langle G', \tau \rangle$-consistency of $W'$. Analogously, if $(W(k), W(j)) \in G, \text{mo} ; G, \text{hb}^\ominus$ for $2 \leq j < k$ then $(W'(f(k)), W'(f(j))) \in G', \text{mo} ; G', \text{hb}^\ominus$, and since $f$ is an increasing function this contradicts the $\langle G', \tau \rangle$-consistency of $W'$. Now, if $(W(k), W(1)) \in G, \text{mo} ; G, \text{hb}^\ominus$, then since $W(1) = w$, $G', \text{E} = G, \text{E} \cup \{e\}$ and $G', \text{rf} = G, \text{rf} \cup \{(w, e)\}$, we have $W'(f(k)) \in \text{dom}(G', \text{mo} ; G', \text{hb}^\ominus ; [G', E^\tau])$, which contradicts the $\langle G', \tau \rangle$-consistency of $W'$.

Lastly, if $W(k) \in \text{dom}(G, \text{mo} ; G, \text{hb}^\ominus ; [E^\tau])$, then $W'(f(k)) \in G', \text{mo} ; G', \text{hb}^\ominus$ (since $e = \max_{\text{max}_{\text{Go}}} G', E^\tau$). However, $W'(f(k)) > \max_{\text{max}_{\text{Go}}} G', E^\tau$, indicating that $W'(f(k)) \in \text{dom}(G', \text{mo} ; G', \text{hb}^\ominus ; [\{W'(j) \mid 1 \leq j < f(k)\}])$, which contradicts the $\langle G', \tau \rangle$-consistency of $W'$. □
**D Effective Pred-Basis**

We show that $P_{\text{loSRA}}$ admits effective pred-basis. For this matter, we demonstrate how to calculate a finite basis $Q^\alpha$ of $\mathcal{P}_{\text{loSRA}}(\uparrow\{B\})$ for each $\alpha$ of the form $(\tau, \mathcal{W}(x, v_\eta))$, $(\tau, \mathcal{R}(x, v_\eta))$, $(\tau, \mathcal{RMW}(x, v_\eta, v_\omega))$ or $\varepsilon$. Then, a finite basis of $\mathcal{P}_{\text{loSRA}}(\uparrow\{B', B''\})$ is given by $\mathcal{P}_{\text{loSRA}}(\uparrow\{B'\}) \times Q^\alpha$ for $\alpha \neq \varepsilon$; and by $\mathcal{P}_{\text{loSRA}}(\uparrow\{B'\})$ for $\alpha = \varepsilon$ (silent memory step). In addition, for a silent program step, a finite basis of $\mathcal{P}_{\text{loSRA}}(\uparrow\{B\})$ is given by $\mathcal{P}_{\text{loSRA}}(\uparrow\{B\})$.

### Silent memory step

The set of predecessors of $B'$ with respect to a silent memory step (i.e., using LOWER) is very simple—it contains any state $B$ such that $B' \subseteq B$. Thus, $\{B'\}$ is a finite basis of $\mathcal{P}_{\text{loSRA}}(\uparrow\{B'\})$, as well as of $\mathcal{P}_{\text{loSRA}}(\uparrow\{B\})$.

### Read

A predecessor $B$ of $B'$ with respect to a read step $B$ is similar to $B'$, except for having in each read-option list of $\tau$ an additional first read option of the form $(\tau, w, v_\eta)$, $\langle \pi, L \rangle$. Hence, for $\alpha = (\tau, R(x, v_\eta))$, the set $\{B'[\tau \mapsto (\tau, w, v_\eta) \cdot B'(\pi)] | \tau_\eta \in \text{Tid}, \tau \in \{R, \text{RMW}\} \}$ is a finite basis of $\mathcal{P}_{\text{loSRA}}(\uparrow\{B'\})$. It is also a basis of $\mathcal{P}_{\text{loSRA}}(\uparrow\{B\})$. For a state $B''$ with $B' \subseteq B''$, a corresponding read option $(\tau, w, v_\eta, u)$ appears in the lists of $\tau$ in $\mathcal{P}_{\text{loSRA}}(\uparrow\{B''\})$ before some additional read options, ensuring that $\mathcal{P}_{\text{loSRA}}(\uparrow\{B'\}) \subseteq \mathcal{P}_{\text{loSRA}}(\uparrow\{B''\})$.

### Write

We construct the basis of the predecessors w.r.t. a write step by considering all (finitely many) possibilities of omitting read options from lists of $B'$, using Prop. C.4 and the following technical lemma:

**Lemma D.1.** Let $P$ be an index choice for $B'$ in loSRA, such that $P \models (\tau, \mathcal{W}(x, v_\eta))$. If $B_0' \subseteq B'$, then $\text{src}(B_0', \tau, P_0) \subseteq \text{src}(B, \tau, P)$ for any index choice $P_0$ for $B_0'$ such that $P_0 \models (\tau, \mathcal{W}(x, v_\eta))$.

**Proof.** Since $B_0' \subseteq B'$, for every $\pi \in \text{Tid}$, there exists a function $f_\pi : B_0'(\pi) \rightarrow B'(\pi)$ such that for every $L_0' \subseteq B_0'(\pi)$, we have $L_0' \subseteq f_\pi(L_0')$, witnessed by a strictly increasing function $f(\pi, L_0') : \{1, ..., |L_0'|\} \rightarrow \{1, ..., |f_\pi(L_0')|\}$, such that $L_0'(k) = (f_\pi(L_0'))(f(\pi, L_0')(k))$ for every $k \in \{1, ..., |L_0'|\}$.

We define $P_0$ to be the positions in $P$ that originated in $B_0'$, according to the $f(\pi, L_0')$ functions. That is, $P_0 = (\lambda \pi \in \text{Tid}, L_0' \subseteq B_0'(\pi), \{k \in \{1, ..., |L_0'|\} | f(\pi, L_0')(k) \in P(\pi, f_\pi(L_0'))\}$.

It is easy to verify that $P_0$ justifies a $(\tau, \mathcal{W}(x, v_\eta))$-step. Let $B_0 = \text{src}(B_0', \tau, P_0)$. We show that $B_0 \subseteq \text{src}(B', \tau, P)$.

Recall that for every thread $\pi \in \text{Tid}$, we have that every list $L_0 \in B_0(\pi)$ is equal to $L_0' \setminus P_0(\pi, L_0')$ (or resp. to $L_0 \setminus P_0(\eta, L_0')$) for some list $L_0'$ of $B_0'(\pi)$ (or resp. for some list $L_0'$ of $B_0'(\eta)$ for some $\eta \in \text{Tid}$). Hence, we can define a function $H_\pi : B_0(\pi) \rightarrow \text{src}(B', \tau, P)(\pi)$, by setting $H_\pi(L_0) = f_\pi(L_0') \setminus P(\pi, f_\pi(L_0'))$. Observe that for every $L_0 \in B_0(\pi)$, we have $L_0 \subseteq H_\pi(L_0)$, witnessed by the function $h(\pi, L_0) : \{1, ..., |L_0|\} \rightarrow \{1, ..., |H_\pi(L_0)|\}$, defined by $h(\pi, L_0)(k) = f_\pi(L_0')(f(\pi, L_0')(k))$, for every $k \in \{1, ..., |L_0|\}$. (Respectively, we define $H_\pi(L_0) = f_\pi(L_0') \setminus P(\pi, f_\pi(L_0'))$, witnessed analogously.)

By Prop. C.4 and Lemma D.1, we get a finite basis of $\mathcal{P}_{\text{loSRA}}(\uparrow\{B\})$, given by:

$$\{\text{src}(B', \tau, P) | P \text{ is an index choice for } B' \text{ such that } P \models (\tau, \mathcal{W}(x, v_\eta))\}.$$

### RMW

The predecessor with respect to an RMW step intuitively combines the predecessors with respect to the read and write steps. By Prop. C.4 and Lemma D.1, we get that the set

$$\{\text{src}(B', \tau, P)[\tau \mapsto (\tau_\eta, x, v_\eta, \text{RMW}) \cdot \text{src}(B', \tau, P)(\pi) | \tau_\eta \in \text{Tid} \text{ and } P \text{ is an index choice for } B' \text{ such that } P \models (\tau, \mathcal{W}(x, v_\eta))\}$$

is a finite basis of $\mathcal{P}_{\text{loSRA}}(\uparrow\{B\})$. 
