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@idc.ac.il

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 a standard model of causal consistency (called also “Causal Convergence” and “Strong-Release-Acquire”). 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.

CCS Concepts: • Software and its engineering → Software verification; Concurrent programming languages; • Theory of computation → Concurrency; Logic and verification; Program verification; • Information systems → Distributed database transactions.

Keywords: weak memory models, causal consistency, release/acquire, shared-memory, concurrency, verification, decidability, well-structured transition systems

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 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., [37] and the mongoDB documentation [40]). 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_acquire and memory_order_release [14, 23, 24]. A stronger form of causal consistency, called SRA (for Strong Release/Acquire), which is equivalent to the standard causal consistency model.
in distributed databases [18]. characterizes the guarantees provided by “multi-copy atomic” multiprocessor architectures, such as POWER. Specifically, as shown in [30], 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. [11] 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 SRA). This implies a non-primitive recursive lower bound on the safety verification problem under causal consistency. Very recently, Abdulla et al. [3] proved that the safety verification problem is undecidable under one instance of causal consistency, namely 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 a (rather tight) 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 use the framework of well-structured transition systems [2, 7, 22]. Intuitively speaking, this framework allows one to establish decidability of infinite-state “lossy” systems, where (i) states may non-deterministically forget some information they include; and (ii) the relation determining whether one state is obtained from another by losing information constitutes a well-quasi-ordering. This approach, however, cannot be applied for (an operationalized version of) SRA directly, whose natural states are execution histories. First, forgetting information from the history results, in many cases, in strictly weaker causality constraints that allow outcomes that cannot be obtained without losing the information. Second, execution histories are only partially ordered and embedding between (general) partial orders is not a well-quasi-ordering.

Our solution is to develop a novel operational semantics that is equivalent to SRA, for which we can use the framework of well-structured transition systems. The key idea in this semantics is to maintain the potential of future reads of each thread in the machine state. This semantics can be straightforwardly made “lossy”, as losing some parts of the possible potential never allows for additional behaviors. In addition, potentials can be represented using total orders, whose embedding relation (based on the ordinary subsequence relation) is a well-quasi-ordering. In this semantics, read transitions are very simple, they only consume a prefix of the potential. 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. This intuition is made precise in our formalized (and mechanized in Coq) correspondence proofs, which establish simulations (forward for one direction and backward for the converse) between the novel 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 [29], where the problem we resolve is left open. As mentioned above, Abdulla et al. [3] proved that safety verification under RA is undecidable. Operational “message-passing” semantics for SRA was developed in [30]. It is inadequate for our purposes as it cannot be made “lossy” without affecting its allowed outcomes.

The safety verification problem was previously investigated 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. [11, 12] establish the decidability of this problem (and the non-primitive recursive lower bound) by reducing it to (and from) reachability 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 behind their reduction cannot be applied for SRA. Notably, SRA cannot be fully explained by program transformations (instruction reordering and merging) [33], whereas, with the exception of the recent undecidability in [3], all existing results (of [12] in particular) are for models that are fully accounted for by such transformations.

More recently, Abdulla et al. [4] 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

---

1This equivalence excludes the atomicity of read-modify-writes, which is crucial in multithreaded programming but is not provided by causal consistency as defined in [18] (see also §3.1).
a global machine memory, while our semantics does not employ any such notion.

Verification of programs under causal consistency (especially under RA) has received considerable amount of attention in recent years. The different approaches include (non-automated) program logics \cite{21, 25, 32, 48, 49}, \cite{21, 25, 32, 48, 49}, (bounded) model checking based on partial order reduction \cite{3, 5, 27, 35} and robustness verification \cite{17, 31, 41}. 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 implementation provides causal consistency guarantees was studied in \cite{16}. 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 a straightforward operational version of SRA’s declarative semantics. In §5 we introduce our novel operational semantics of SRA. In §6 we show how this semantics is used to decide the safety verification problem. We conclude in §7. The appendices to this paper, publicly available in \cite{1}, provide full proofs. Mechanized Coq proofs of the equivalence of the two semantics of SRA are available in the artifact accompanying this paper.

## 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 systems with execution graphs (§2.3).

### 2.1 Programming Language

Let Val ⊆ \(\mathbb{N}\), Loc ⊆ \(\{x, y, \ldots\}\), Reg ⊆ \(\{a, b, \ldots\}\) 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 \(r\) (invoking no memory access); if \(e\) go to \(n\) 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 \(r := \text{CAS}(x, e_r, e_w)\) atomically loads the value of \(x\) to \(r\), compares it to the value of \(e_r\), and if the two values are equal, replaces the value of \(x\) by the value of \(e_w\).

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 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 \(\text{Tid} ⊆ \{T_1, T_2, \ldots\}\) of thread identifiers to 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 6).

### 2.2 From Programs to Labeled Transition Systems

Sequential and concurrent programs induce labeled transition systems.

**Labeled transition systems.** A labeled transition system (LTS) \(A\) over an alphabet \(\Sigma\) is a triple \((Q, Q_0, T)\), where \(Q\) is a set of states, \(Q_0 \subseteq Q\) is the set of initial states, and \(T \subseteq Q \times \Sigma \times Q\) is a set of transitions. We denote by \(A, Q_0\) and \(A, T\) the components of an LTS \(A\); write \(\sigma \rightarrow_A \tau\) for the relation \(\{(q, q') | (q, \sigma, q') \in A\}\) and \(\rightarrow_A \cup_{\sigma \in \Sigma} \rightarrow_A\). A state \(q \in A, Q\) is reachable in \(A\) if \(q_0 \rightarrow_A q\) for some \(q_0 \in A, Q_0\). A sequence \(\sigma_1, \ldots, \sigma_n\) is a trace of \(A\) if \(q_0 \stackrel{\sigma_1}{\rightarrow}_A \cdots \stackrel{\sigma_n}{\rightarrow}_A q\) for some \(q_0 \in A, Q_0\) and \(q \in A, Q\). The set of pre-decessors of a set \(S \subseteq A, Q\) w.r.t. a symbol \(\sigma \in \Sigma\), denoted by \(\text{pred}_{\sigma}(S)\), is given by \(\{q \in A, Q | \exists q' \in S. q \stackrel{\sigma}{\rightarrow}_A q'\}\). We define \(\text{pred}_{\epsilon}(S) \triangleq \bigcup_{\sigma \in \Sigma} \text{pred}_{\sigma}(S)\).

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 \(R(x, v_r)\) (read label), \(W(x, v_w)\) (write label) or \(\text{RMW}(x, v_r, v_w)\) (read-modify-write label), where \(x \in \text{Loc}\) and \(v_r, v_w \in \text{Val}\). We denote by \(\text{Lab}\) the set of all labels. The functions \(\text{typ}\), \(\text{loc}\), \(\text{val}_{\text{R}}\), and \(\text{val}_{\text{W}}\) return (when applicable) the type (\(R/W/\text{RMW}\)), location, read value and written value of a given label \(l\).

A sequential program \(S\) ∈ 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 \((0, \lambda r \in \text{Reg}, 0)\) 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 LTTSs (when writing, e.g., \(S, Q\) and \(\rightarrow_S\)).

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 \(\overline{P}\), assigning a state in \(P(r, Q)\) to every \(r \in \text{Tid}\); its initial states set is \(\{\overline{P} | \forall r. \overline{P}(r) \in P(r, Q_0)\}\); and its transitions are
\(v \in \text{Val} \subseteq \mathbb{N}\)
\(x, y, z \in \text{Loc} \subseteq \{x, y, \ldots\}\)
\(r \in \text{Reg} \subseteq \{a, b, \ldots\}\)
\(r, \tau, \eta \in \text{Tid} \subseteq \{T_1, T_2, \ldots\}\)
\(S \in \text{SProg} \equiv \{0, 1, \ldots, N\} \rightarrow \text{Inst}\)
\(P : \text{Tid} \rightarrow \text{SProg}\)

_Values, locations, registers, thread identifiers, sequential programs, (concurrent) programs_

\(e \equiv r \mid v \mid e + e \mid e = e \mid e \neq e \mid \ldots\)

**Figure 1.** Domains, metavariables and programming language syntax.

\(S(pc) = r := e\)
\(S(pc) = \text{if } e \text{ goto } n\)
\(S(pc) = \text{if } e \text{ goto } n\)
\(S(pc) = l := e\)
\(S(pc) = e := e\)

**Figure 2.** Transitions of LTS induced by a sequential program \(S \in \text{SProg}.\)

"interleaved transitions" of \(P\)’s components, given by:

\[
\begin{align*}
\text{l} \in \text{Lab} & \quad p \xrightarrow{l_1} p' \xrightarrow{l_2} p'' \quad \xrightarrow{\text{if} } p \xrightarrow{l_1} p' \xrightarrow{l_2} p'' \quad \xrightarrow{\text{else} } p \xrightarrow{l_1} p' \xrightarrow{l_2} p''
\end{align*}
\]

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 \(e = \langle \tau, n, l \rangle\), where \(\tau \in \text{Tid}\) 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 \(\text{typ}, \text{loc}, \text{val}_x, \text{val}_{ax}\) 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 \(\text{RMW} \equiv \mathbb{R} \cap \text{W}\). Sub/superscripts are used to restrict these sets to certain location (e.g., \(W_x = \{w \in W \mid \text{loc}(w) = x\}\)) and/or thread identifier (e.g., \(E^\tau = \{e \in E \mid \text{tid}(e) = \tau\}\)).

Our representation of events induces a partial order \(\subseteq\) on them: events of the same order are ordered according to their serial numbers (i.e., \(\langle r_1, n_1, l_1 \rangle \subseteq \langle r_2, n_2, l_2 \rangle\) 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 \(rf\) is a reads-from relation for a set \(E\) of events if the following hold:

- If \(⟨w, r⟩ \in rf\), then \(w \in E \cap W, r \in E \cap R, \text{loc}(w) = \text{loc}(r)\) and \(\text{val}_x(w) = \text{val}_x(r)\).
- If \(⟨w_1, r⟩, ⟨w_2, r⟩ \in rf\), then \(w_1 = w_2\) (that is, \(rf^{-1} = \{⟨r, w⟩ \mid ⟨w, r⟩ \in rf\}\) is functional).
- \(∀r \in E \cap W. \exists w. ⟨w, r⟩ \in rf\) (each read event reads from some write event).

**Definition 2.4.** A relation \(mo\) is a modification order for a set \(E\) of events if \(mo_{x} \in \text{Loc}\) where each \(mo_x\) is a strict total order on \(E \cap W_x\).

**Definition 2.5.** An execution graph is a triple \(G = \langle E, rf, mo \rangle\) where \(E\) is a finite set of events, \(rf\) is a reads-from relation for \(E\) and \(mo\) is a modification order for \(E\). We denote by \(E\text{Graph}\) the set of all execution graphs. The components of \(G\) are denoted by \(G.E, G.rf, G.mo\), and \(G.po\) denotes the restriction of \(<\) to \(E.G\) (i.e., \(G.po \equiv \{e_1, e_2) \in E \times E \mid e_1 < e_2\}\)). For a set \(E \subseteq E\), we write \(G.E\) for \(G.E \cap E\) (e.g., \(G.W_x = G.E \cap W_x\)).

The next definition is used to associate execution graphs to programs. Multiple examples below (on Page 6) illustrate execution graphs of different programs.

**Notation 2.6.** For a set \(E\) of events, thread identifier \(\tau \in \text{Tid}\) and label \(l \in \text{Lab}, \text{NextEvent}(E, r, l)\) denotes the event given by \(\langle τ, 1 + \max\{n \in \mathbb{N} \mid \exists l' \in \text{Lab.} \langle τ, n, l' \rangle \in E\}\rangle, l\rangle\).

**Definition 2.7.** An execution graph \(G\) is generated by a program \(P\) with final state \(Õ\) if \(p_0 \in P.Q_0\), where \(G_0\) denotes the empty execution graph (given by \(G_0 = \langle E, 0, 0, 0\rangle\) and \(\rightarrow\) is defined by:

\[
\begin{align*}
\text{if } & \quad p_0 \xrightarrow{\text{rf}} p' \quad E' = E \cup \text{NextEvent}(E, r, l) \quad rf \subseteq rf' \quad mo \subseteq mo' \quad \text{then } \quad p_0 \xrightarrow{rf'} p' \quad (G, E \rightarrow \langle p', G' \rangle) \quad \text{is a valid transition.}
\end{align*}
\]
3 The Strong Release/Acquire Model

Declarative memory models, such as Strong Release/Acquire (SRA), are formulated by a collection of constraints on execution graphs, which determine the consistent execution graphs—the ones allowed by the model. In this section, we formulate the constraints of SRA and define the safety verification problem under SRA, discuss equivalent alternative formulations (§3.1), provide several examples (§3.2), and investigate the relation between SRA and RA (§3.3).

Notation 3.1 (Relations). Given a relation $R$, $\text{dom}(R)$ denotes its domain; $R'$ and $R^*$ denote its reflexive and transitive closures; and $R^{-1}$ denotes its inverse. The (left) composition of relations $R_1; R_2$ is denoted by $R_1 \circ R_2$. We denote by $[A]$ the identity relation on a set $A$, and so $[A] \circ R \circ [B] = R \cap (A \times B)$.

Causal consistency models are based on the following basic derived “happens-before” relation:

$$ G_{hb} \triangleq (G_{po} \cup G_{rf})^+ $$

The happens-before relation captures the “causality relation” in execution graphs. In words, $hb$ is the smallest transitive relation that contains the program order ($po$) and the reads-from ($rf$) relations. We note that all writes synchronize with the writes they read from ($rf \subseteq hb$), in contrast to more elaborate models like RC11 [34], where only certain reads from edges induce synchronization.

Given $hb$, 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:

\[ \text{irr-hb-mo} \]

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 [39], which, unless restricted appropriately, lead to the infamous “out-of-thin-air” problem [13, 26].

\[ \text{read-coherence} \]

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 $r$ being aware of some write event $w$” with an $hb$-path from $w$ to (some event of) $r$, and using the modification order $mo$ to interpret one write being “later” than another, the precise condition requires that:

$$ G_{mo} ; G_{hb} ; G_{rf}^{-1} \text{ is irreflexive} \quad \text{(read-coherence)} $$

Indeed, if a read event $r$ reads from a write event $w_1$, while being aware of an $mo$-later write event $w_2$ to the same location, we have $\langle w_1, w_2 \rangle \in mo$, $\langle w_2, r \rangle \in hb$ and $\langle r, w_1 \rangle \in rf^{-1}$.

\[ \text{atomicity} \]

This condition ensures that RMWs are stronger than a read followed by a write. It requires that RMWs read from their immediate $mo$-predecessors:

$$ G_{mo} ; G_{mo} ; G_{rf}^{-1} \text{ is irreflexive} \quad \text{ (atomicity)} $$

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 $\overline{p}$ of a program $P$ reachable under SRA if some SRA-consistent execution graph is generated by $P$ with final state $\overline{p}$ (see Def. 2.7).

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

\[ \text{Input:} \text{ a program } P \text{ and a “bad” state } \overline{p} \in P.Q. \]

\[ \text{Question: is } \overline{p} \text{ reachable under SRA?} \]

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. [11] to safety verification under x86-TSO.

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

3.1 Other Formulations of SRA

Our presentation above follows [30], where SRA is introduced as a strengthening of RA. The latter is the fragment of the C/C+++11 model [14, 34] consisting of release stores, acquire reads and acquire-release RMWs. In addition, SRA appears (in multiple disguises) in the literature:

\[ \text{POWER.} \] As proved in [30], SRA precisely coincides with the POWER model of [9], 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 [38] (that is, placing $\text{lwsync}$ before every store and $\text{ctrl+isync}$ after every load).

\[ \text{Distributed Key-Value Stores.} \] Ignoring RMWs, the SRA model is equivalent to the causal convergence model, denoted by CCv, of [16] (when applied to the standard read/write memory sequential specification), as well as to the causal consistency model of [37] when restricted to single-instruction transactions. These models are formulated in [18, 20] in terms of visibility ($vis$) and arbitration ($ar$) relations. One direction of the correspondence follows by setting $vis = hb$ and taking $ar$ to be some total order extending $hb \cup mo$. For the converse, one takes $rf$ to relate each read $r$ with the $ar$-maximal write to the same location that is $vis$-before $r,$
and sets $\text{mo} = \bigcup_{x \in \text{Loc}} \{W_x \mid a_1 \vdash W_x \}$. Furthermore, our program order (po) corresponds to session order (so), and SRA’s consistency ensures strong session guarantees (so $\subseteq$ vis) [47].

RMWs in distributed databases require expensive global coordination. A naive implementation of RMWs as transactions that read and write from/to the same location does not guarantee atomicity, as it allows the lost update anomaly (e.g., it will allow the outcome in Ex. 3.9 below). In the particular case when a certain 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 [15, 36].

**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) [10, 15, 19, 43, 45] 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 well-known litmus tests to demonstrate SRA (some of which are revisited in the sequel). To simplify the presentation, instead of referring to reachable program states, we consider possible program outcomes assigning final values to (some) registers. An outcome $O : \text{Reg} \rightarrow \text{Val}$ is allowed for a program under the declarative model SRA if some state in which the registers are assigned their values in $O$ is reachable under SRA (see Def. 3.2). We use program comment annotations (“//”) to denote particular outcomes.

**Remark 1.** To simplify our presentation, we require explicit initialization of memory locations and adapt the examples to include explicit initialization. Reading from an uninitialized location blocks the thread. (For example, only the initial execution graph $G_0$ is generated by a program consisting of a single thread that reads from some location, without previously writing to it.) This is only a presentation matter: one may always achieve implicit initialization by augmenting the program with an additional thread that sets all variables to their initial value, and then signals all other thread (using an additional flag) to start running.

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

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

In its execution graph the $\text{rf}$-edges are forced because of the read values, whereas the $\text{mo}$-edges are forced due to $\text{irr-hb-mo}$. 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 outcome is disallowed:

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

An execution graph for this outcome must have $\text{rf}$ and $\text{mo}$-edges as depicted above. However, we have $\text{mo}$ from $W(x, 0)$ to $W(x, 1)$, $\text{hb}$ from $W(x, 1)$ to $R(x, 0)$ and $\text{rf}$ 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 (Transitive message passing).** $\text{po}$ and $\text{rf}$ edges equally contribute to $\text{hb}$ in causal consistency. Hence, as in Ex. 3.6, the following outcome is disallowed by SRA.

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

**Example 3.8 (Independent reads of independent writes). A main difference between SRA and the x86-TSO model [42] 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 $W(x, 1)$.

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

**Example 3.9.** For the implementation of locks, it is crucial that two RMWs never read from the same write:

\[
\begin{align*}
x &:= 0 \quad a &:= y //1 \\
x &:= 1 & b &:= x //0 \\
\text{CAS}(x, 0, 1) &\not\mid \text{CAS}(x, 0, 1) \\
\text{SRA} & & \text{2RMW} \\
\end{align*}
\]

Since $\text{mo}$ must order the two RMWs and $\text{irr-hb-mo}$ dictates that $\text{mo} ; \text{rf}$ is irreflexive, any order of the RMWs entails a violation of atomicity.

**Example 3.10.** RMWs to an otherwise-unused location can be used as fences, as the consistency constraints imply that $\text{hb}$ must totally order $G.W_x$ 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.

\[
\begin{align*}
z &:= 0 \\
x &:= 0 \\
x &:= 1 \\
a &:= \text{FADD}(z, 0) \\
b &:= y//0 \\
y &:= 0 \\
y &:= 1 \\
c &:= x//0
\end{align*}
\]

\[
\begin{align*}
W(z, 0) & \downarrow \\
W(x, 0) & \downarrow \\
W(y, 0) & \downarrow \\
W(x, 1) & \downarrow \\
W(y, 1) & \downarrow \\
R(y, 0) & \downarrow \\
R(x, 1) & \downarrow \\
\end{align*}
\]

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:

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

- G.hb is reflexive (irr-hb)

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

- G.mo; G.hb is reflexive (write-coherence)

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

Example 3.11. 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 (of SRA) restricts the relation between [W_x]; mo; [W_y] and [W_y]; mo; [W_y] also when x ≠ y. The following example (adapted from [50]) demonstrates the difference:

\[
\begin{align*}
x &:= 1 \\
y &:= 1 \\
x &:= x//2 \\
a &:= y//1 \\
\checkmark & \text{RA} \\
\end{align*}
\]

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 3.12. 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 [8], 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.

Definition 3.13. An execution graph G is write/write-race free if for every w_1, w_2 ∈ G.W with \text{lloc}(w_1) = \text{lloc}(w_2), we have w_1 = w_2, (w_1, w_2) ∈ G.hb or (w_2, w_1) ∈ 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 3.14. 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. 3.12, it suffices to show that reachability under RA implies reachability 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 an execution graph of the form \{E_p, [E_p] ; G.rf ; [E_p] ; G.mo ; [E_p]\} where E_p ⊆ G.E and dom(G.hb ; [E_p]) ⊆ E_p.) Since the empty execution graph G_0 is trivially SRA-consistent, G cannot be empty. Let e be a G.hb-maximal event in G.E, and let E' = G.E \ {e}. The minimality of G ensures that G' = (E', [E'] ; G.rf ; [E'] ; G.mo ; [E']) (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 G'.mo ⊆ G'.hb ⊆ G.hb.

Now, since G is RA-consistent but not SRA-consistent, 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 (e, w) ∈ G.mo and (w, e) ∈ (G.hb \cup G'.mo)⁺. Since G'.mo ⊆ G.hb, it follows that (e, e) ∈ G.mo \ G.hb. Hence, G does not satisfy write-coherence, which contradicts the fact that it is RA-consistent. □

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 $\text{Lab}$, representing the thread identifier and the label of the operation, or $e$ for internal (silent) memory actions.

Example 4.2. The most well-known memory system is the one of sequential consistency, denoted here by $\text{SC}$. This memory system simply tracks the most recent value written to each location (or $\bot$ for uninitialized locations). Formally, it is defined by $\text{SC.Q} \triangleq \text{Loc} \rightarrow \{\bot\}$. The states set is given by:

$$\mu' = \mu[x \mapsto \text{val}_\tau(x)]$$

Note that $\text{SC}$ is oblivious to the thread that takes the action ($\mu \xrightarrow{r.l}_{\text{SC}} \mu'$ iff $\mu \xrightarrow{r.l}_{\text{SC}} \mu'$), and it has no silent transitions.

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

Definition 4.3. A program $P$ and a memory system $M$ form a concurrent system, denoted by $(P, M)$. It is an LTS over $(\text{Tid} \times (\text{Lab} \cup \{e\}) \cup \{e\})$ whose set of states is $P, Q \times M, Q_0$; its initial states set is $P, Q_0 \times M, Q_0$; and its transitions are "synchronized transitions" of $P$ and $M$, given by:

$$l \in \text{Lab} \quad \tilde{p} \xrightarrow{r.l} p' \quad m \xrightarrow{r.l}_M m'$$

Next, we present the memory system $\text{opSRA}$ that is equivalent to $\text{SRA}$ (in the sense that is made formal in Thm. 4.5). We also refer the reader to Fig. 5 on Page 12, which illustrates a run of $\text{opSRA}$ 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 memory graph $G_0\text{(opSRA.Q}_0 \triangleq \{G_0\}$; and the transitions are given in Fig. 3. A $\text{WRITE}$ step by thread $r$ adds a corresponding fresh write event $e$ to the graph placed after all events of thread $r$ and extends $\text{mo}$ to order $e$ after all existing writes to the same location. A $\text{READ}$ step by thread $r$ 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 ($\text{val}_\tau(w) = \text{val}_\tau(x)$), and the thread executing the read is not aware of an $\text{mo}$-later write to the same location ($w \notin \text{dom}(\text{mo}; h^B_m; [E'])$). 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 $\text{SRA}$-consistent execution graphs (as per $\text{irr-hb-mo}$). Hence, to generate an $\text{SRA}$-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 $\text{SRA}$ and $\text{opSRA}$.

Definition 4.4. A state $\tilde{p}$ of a program $P$ is reachable under a memory system $M$ if $\tilde{p}$ is reachable in $P, M$ for some $m \in M, Q$.

Theorem 4.5. A state $\tilde{p}$ of program $P$ is reachable under $\text{SRA}$ (see Def. 3.2) if it is reachable under $\text{opSRA}$.

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

Remark 2. Following [27], our formulation of $\text{opSRA}$ does not directly refer to the consistency predicates, but rather articulate necessary and sufficient conditions that ensure that the target state is a consistent execution graph. It is possible to take a step further and develop an equivalent semantics with more compact states that may feel "more operational" and intuitive. Indeed, it suffices to maintain a partially ordered set of write events, together with a mapping of which writes each thread is aware of (the "observed writes set" of [21]). This can be implemented using timestamps, messages and thread views, as was done, e.g., in [25].
5 Making Strong Release/Acquire Lossy

For resolving the reachability problem under SRA, we introduce an alternative memory system, which we call loSRA (for “lossy-SRA”). In this section, we present loSRA, 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 loSRA maintains a collection of “read-option” lists for each thread, called the potential of the thread, where each read option o contains a location loc(o), a value val(o) 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 o1 · o2 allows the thread to read val(o1) from location loc(o1) and then val(o2) from location loc(o2). 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 (recall that we assume explicit initialization, see Remark 1). 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 τ to read v from x, an option o with val(o) = v and loc(o) = x must be the first in each of τ’s lists. Then, to progress to the next option in the list, the thread may consume these options, and discard the first element from each of its lists.

A write step is more involved, encapsulating the requirements of opSRA. First, since opSRA performs write events following their 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). Accordingly, we do not allow a thread to write to x if some read option o with loc(o) = 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 o with loc(o) = x and val(o) = v may be added to every list of every thread. This makes the write step in loSRA (unlike the one of opSRA) non-deterministic—the writer essentially has to “guess” what thread will read from the new write and when.

But, where in the lists should we allow to add such options? The following examples demonstrate two possible cases. We write in them o x v for a read option of value v from location x.

Example 5.1. Consider the IRIW program with its (SRA-allowed) outcome in Ex. 3.8. Clearly, the first step may only be a write by T1 or T4. Suppose, w.l.o.g., that T1 begins. Since T3 reads 0 from x, a read option o x 0 should be added in the lists of T3. Now, before reading 0 from x, T3 has to read 1 from y. Hence, when T4 writes 1 to y, a read option o y 1 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 1 in the list of T1 placed before o x 0. Finally, T2 may run: read 1 from y (consuming o x 0) and then 0 from x (consuming o y 1).

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

Shared-memory causality principle: After thread τ 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 τ 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 Example 5.2, the read option o y 1 cannot be placed before o x 0, because T1 cannot have o y 1 in its lists at the point of writing 1 to y.)

Now, since the potential of thread τ is used both for (i) dictating future reads of τ, and (ii) 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:

```
x := 0
x := 1
y := 0
y := 1
a1 := z = 0
b1 := x = 0
c1 := y = 0
d1 := x = 0
e1 := y = 0
f1 := z = 0
a2 := y = 0
b2 := z = 0
c2 := x = 0
d2 := z = 0
e2 := z = 0
f2 := x = 0
```

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 y 1 and o x 0 in this order. In addition, a read option o y 1 should be placed in T3’s list before o x 0. The justification for it requires o y 1 · o x 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 1 and o y 1 · o x 0. But, no such interleaving is a possible future for T3 (and thus cannot be generated by loSRA): reading o y 1 does not allow T3 to read

\[A weaker observation, which only considers single reads, was essential for the soundness of OGRA—an Owicki Gries logic for RA introduced in [32].\]
relation $\subseteq$ 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: $B \sqsubseteq B'$ if $B(\tau) \subseteq B'(\tau)$ for every $\tau \in$ Tid.

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

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

To achieve implicit initialization of all locations to 0, one should take IoSRA, $Q_0$ consist of all functions assigning to each thread sequences consisting of read options of the form $(T_0, x, 0, u)$ where $T_0$ is a distinguished thread identifier that is not used in programs (corresponds to the initializing thread, see Remark 1).
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: (i) the suffix of the existing list right after the position of the first added option is a read-option list of the writing thread; (ii) 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 (iii) 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 (iii).

**Remark 3.** Our formal write step 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.

**Remark 4.** Our formal read step always discards the first option from the lists, which was used to justify the read. An alternative semantics that keeps the lists unchanged in read steps (allowing to discard the first option using the lower step) would be completely equivalent. Indeed, the write step that added the consumed option could always add multiple identical consecutive read options.

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 3). Intuitively, lower can only reduce the possible traces, while it allows us to show that IoSRA is a well-structured transition system.

**Example 5.9.** Consider the $2+2W$ program with its (SRA-disallowed) outcome in Ex. 3.11. To see that this outcome cannot be obtained by IoSRA, 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. 3.9. To try to obtain this outcome in IoSRA, the $x := 0$ by $T_1$ must add a read option $(\pi_1,x,0,RMW)$ 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 read 0, and this outcome cannot be obtained by IoSRA.

Next, we establish the equivalence of IoSRA and opSRA. To do so, we define a relation $\forall \subseteq IoSRA \times opSRA$, formalizing the intuitive simulation discussed so far between IoSRA’s lists and opSRA’s execution graphs. For defining $\forall$, 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 $L(k) = \langle \tau,x,v,u \rangle$:

- $W(k) \in G.W$.
- $\text{tid}(W(k)) = \tau$, $\text{loc}(W(k)) = x$ and $\text{val}_{\text{w}}(W(k)) = v$.
- 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 \( \text{mo} \)-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 \( \text{mo} \)-later than some write \( w_2 \) that appears after \( w_1 \) in \( W \). Formally:

**Definition 5.12.** A write list \( W \) is called \( \langle G, \tau \rangle \)-consistent if \( W(k) \not\in \text{dom}(G, \text{mo}; G, \text{hb}^?) \) for every \( 1 \leq k \leq |W| \).

Now, \( \forall \) relates a \( \text{loSRA} \) state \( B \) with an execution graph \( G \) if each read-option list in \( B \) has an appropriate write list.

**Definition 5.13.** A state \( B \in \text{loSRA}, Q \) matches an execution graph \( G \), denoted by \( B \vdash G \), if for every \( \tau \in \text{Tid} \) and \( L \in B(\tau) \), there exists a \( \langle G, \tau \rangle \)-consistent \( \langle G, L \rangle \)-write-list.

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

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

Full proofs of Lemmas 5.15 and 5.16 are provided in [1], and their mechanization in the Coq proof assistant is provided in the artifact accompanying this paper.

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

**Proof Outline.** We show that \( \forall \) constitutes a (weak) forward simulation from \( \text{loSRA} \) to \( \text{opSRA} \). Handling the lower trace is easy (new write lists are restrictions of the ones we have).

Now, suppose that (i) \( B \vdash G \), witnessed by a \( \langle G, \pi \rangle \)-consistent \( \langle G, L \rangle \)-write-list \( W(\pi, L) \) for every \( \pi \in \text{Tid} \) and \( L \in B(\pi) \); and (ii) \( B \overset{\tau, \pi}{\longrightarrow}_{\text{opSRA}} B' \). We construct \( G' \) such that \( B' \vdash G' \) and \( G' \overset{\tau, \pi}{\longrightarrow}_{\text{opSRA}} G' \) (as depicted on the right).

We describe here the write and read steps (the \text{rmw} 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, \( G' \overset{\tau, \pi}{\longrightarrow}_{\text{opSRA}} G' \) is trivial. To show that \( B' \vdash G' \), we construct for every \( \pi \in \text{Tid} \) and \( L' \in B'(\pi) \), a \( \langle G', \pi \rangle \)-consistent \( \langle G', L' \rangle \)-write-list \( W' \). The write list \( W' \) maps: (i) the new options to the new write event \( w \); (ii) the "old" options that appear before the first new option as mapped in the existing write list \( W \) for the corresponding list in \( B(\pi) \); and (iii) each old option that appears after the first new option to the \text{mo}-maximal write event among (1) its mapping in the existing write list \( W \) for the corresponding list in \( B(\pi) \) and (2) its mapping in the existing write list \( W_0 \) for the justifying list in \( B(\tau) \). Roughly speaking, picking the \text{mo}-maximal write in the third case ensures that \( W' \) is \( \langle G', \pi \rangle \)-consistent: In the new state, thread \( \pi \) 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 between these two writes.

In turn, to simulate a read step of \( \text{loSRA} \) in \( \text{opSRA} \), 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 \( L \in B(\tau) \). (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 \( w \not\in \text{dom}(G, \text{mo}; G, \text{hb}^?) \) for \( \text{E}^? \), so we can make
the read step in opSRA when reading from w. To show that B′ ∪ G′, for every thread π ≠ τ, we simply reuse the write list W(π, L), that we had for G; while for thread τ itself, we shift its write lists by one (setting W′(r, L) = λk. W(r, L) + 1), and use the po-minimality of w to establish ⟨G′, τ⟩-consistency. □

For the converse, we favor backward simulation, since IoSRA 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 IoSRA.

**Proof Outline.** We show that γ⁻¹ constitutes a backward simulation from opSRA to IoSRA.

For the main proof obligation (depicted on the right), suppose that G ⊢ τ opSRA G′ and B′ ⊆ G′, witnessed by a (G′, π)-consistent ⟨G′, L′⟩-write-list W(π, L′), for every π ∈ Tid and L′ ∈ B′(π). We construct a state B such that B ⊢ τ opSRA B′ and B ⊇ G′.

Again, we describe here the write and read steps. First, for a write step, let w be the write event that is added in opSRA’s transition from G to G′. Roughly speaking, we construct B by removing the read options that were associated to w in the existing write lists, and copying the suffix of each read-option list after the first read option associated to w to the potential of thread τ. The write lists for B are then induced by those for B′ in the obvious way.

In turn, for a read step, let r be the read event that is added in opSRA’s transition from G to G′, and w be the write event that r reads from in G′. Then, we construct B by setting B = B′[r → ⟨tid(w), loc(r), vala(r), R⟩, B′(r)], and B ⊢ τ opSRA B′ follows by definition. Now, to show that B ⊇ G, we use the write lists of B′ for every π ≠ τ. For π = τ we append w in the beginning of the write lists of B′. □

We conclude with the equivalence of opSRA and IoSRA.

**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 IoSRA.

**Proof.** Directly follows from Lemmas 5.15 and 5.16. □

6 Decidability of the Reachability Problem

We show IoSRA 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.

**Preliminaries.** A well-quasi-ordering (wqo) on a set S is a reflexive and transitive relation ≤ on S such that for every infinite sequence s₁, s₂, ... of elements of S, we have s_i ≤ s_j for some i < j. In a context of a set S and a wqo ≤ on S, the upward closure of a set U ⊆ S, denoted by ↑U, is given by {s ∈ S | ∃u ∈ U. u ≤ s}; a set U ⊆ S is called upward closed if ↑U = U; and a set B ⊆ U is called a basis of U if U = ↑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 ≤ on A.Q that is compatible with A, that is: if q₁ →ₐ q₂ and q₁ ≤ q₃, then there exists q₄ ∈ A.Q such that q₃ →ₐ q₄ and q₂ ≤ q₄. The coverability problem for ⟨A, ≤⟩ asks whether an input state q ∈ A.Q is coverable, namely: is there some q′ with q ≤ q′ reachable in A?

Coverability is decidable (see, e.g., [7, 22]) for a WSTS ⟨A, ≤⟩ provided that ≤ is decidable and the following hold:

(i) **effective initialization:** there exists an algorithm that accepts a state q ∈ A.Q and decides whether ↑q ∩ A.Q = ∅.
(ii) **effective pred-basis:** there exists an algorithm that accepts a state q ∈ A.Q and returns a finite basis of ↑predₐ ↑q.

Roughly speaking, these conditions ensure that (i) backward reachability analysis from q will converge to a fixed point; (ii) each step in its calculation is effective; and (iii) we can check whether the fixed point contains an initial state.

**IoSRA as a Well-Structured Transition System.** The ⊆ ordering on the states of IoSRA is clearly decidable and also forms a wqo. Indeed, by Higman’s lemma, ⊆ is a wqo on the set of all read-option lists. In turn, its lifting to potentials (which are finite by definition) is a wqo on the set of all potentials (see [44]). Finally, by Dickson’s lemma, the pointwise lifting of ⊆ to functions assigning a potential to every τ ∈ Tid (i.e., states of IoSRA) is also a wqo.

Now, let P be a program. The ⊆ ordering is naturally lifted to states of the concurrent system P_{IoSRA} (that is, pairs (, B) ∈ P.Q × IoSRA.Q, see Def. 4.3) by defining (, B) ⊆ (, B′) iff = and B ⊆ B′.

**Lemma 6.1.** P_{IoSRA} equipped with ⊆ is a WSTS that admits effective initialization and effective pred-basis.

**Proof.** First, since P.Q is (by definition) finite and ⊆ is a wqo on IoSRA.Q, we have that ⊆ is a wqo of P_{IoSRA}.Q.

Second, since lower is explicitly included in IoSRA, ⊆ is clearly compatible with P_{IoSRA}. Indeed, given q₁ = (₁, B₁), q₂ = (₂, B₂) and q₃ = (₃, B₃) such that q₁ →₁ P_{IoSRA} q₂ and q₁ ⊆ q₃ (so ₁ = ₃), for q₂ = q₃, we have q₃ →ₐ P_{IoSRA} q₄ (since B₃ ∈ P_{IoSRA} B₁ using the lower step) and q₃ ⊆ q₄.

Next, P_{IoSRA} trivially admits effective initialization. Indeed, the states (, B) for which ↑(, B) ∩ P_{IoSRA}.Q₀ ≠ ∅ are exactly the initial states themselves—P.Q₀ × {ε}.

Finally, [1] establishes the effective pred-basis for P_{IoSRA}. For this matter, we demonstrate how to calculate a finite basis of ↑predₐ P_{IoSRA}(↑B) for α of the form (τ, W(x, vₐ)), (τ, R(x, vₐ)), (τ, RMN(x, vₐ, vₐ)) or ε. □

It is now easy to establish the decidability of reachability under IoSRA.
Theorem 6.2 (loSRA reachability). Given a program $P$ and a state $\bar{p} \in P.Q$, it is decidable to check whether $\bar{p}$ is reachable (see Def. 4.4) under the memory system loSRA.

Proof. Since the first component (the program state) in $\subseteq$-ordered pairs of $P_{\text{loSRA}}$’s states is equal, reachability under loSRA is reduced to coverability in $(P_{\text{loSRA}}, \subseteq)$, which is decidable by Lemma 6.1 and the framework of [7]. □

We are now in position to prove our main results.

Corollary 6.3. The SRA reachability problem is decidable.

Proof. Directly follows from Theorems 4.5, 5.17 and 6.2. □

Corollary 6.4 (RA race-free reachability). Given a program $P$ that is write/write-race-free under SRA (see Def. 3.13) and a state $\bar{p} \in P.Q$, it is decidable to check whether $\bar{p}$ is reachable under RA.

Proof. Directly follows from Thm. 3.14 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. [4] 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 [3]. 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. 3.14), our result allows the verification of safety properties under RA for this large and widely used class of programs. Concurrent separation logics [25, 48, 49], designed for verification under RA, are also essentially limited to reason only about write/write-race-free programs and stateless model checking is significantly simpler with this assumption (see [27, §5]). We also note that it is straightforward to support C/C++11’s non-atomic, 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. 3.10). Indeed, as demonstrated in [25], it suffices to check for data races assuming RA semantics. The extension to other fragments of C/C++11, 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., [28, 46]). Other directions for future work include handling other variants of causally consistent shared-memory (see, e.g., [16]), 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 [4, 6]).

Acknowledgments

We thank the PLDI’20 reviewers for their helpful feedback. This research was supported by the Israel Science Foundation (grants 5166651 and 1373/16), and by Len Blavatnik and the Blavatnik Family foundation. The first author was also supported by the Alon Young Faculty Fellowship.

References


