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

ACM Reference Format:
Ori Lahav and Udi Boker. 2020. Decidable Verification under a Causally Consistent Shared Memory. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '20), June 15–20, 2020, London, UK*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3385412.3385966

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_release` and `memory_order_acquire` [14, 23, 24]. A stronger form of causal consistency, called SRA (for Strong Release/Acquire), which is equivalent to the standard causal consistency model

PLDI '20, June 15–20, 2020, London, UK

ACM ISBN 978-1-4503-7613-6/20/06.
https://doi.org/10.1145/3385412.3385966

in distributed databases [18],[1] 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 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

[1] This 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 [21, 25, 32, 48, 49], (bounded) model checking based on partial order reduction [3, 5, 27, 35] and robustness verification [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 [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 [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 $\mathsf{Val} \subseteq \mathbb{N}$, $\mathsf{Loc} \subseteq \{\mathtt{x}, \mathtt{y}, \dots\}$, $\mathsf{Reg} \subseteq \{\mathtt{a}, \mathtt{b}, \dots\}$ 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); $\mathtt{if}\ e\ \mathtt{goto}\ n$ 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; $r := \mathtt{FADD}(x, e)$ atomically increments x by the value of e and loads the old value of x to r; $r := \mathtt{XCHG}(x, e)$ atomically swaps x to the value of e and loads the old value of x to r; and $r := \mathtt{CAS}(x, e_{\mathtt{R}}, e_{\mathtt{W}})$ atomically loads the value of x to r, compares it to the value of $e_{\mathtt{R}}$, and if the two values are equal, replaces the value of x by the value of $e_{\mathtt{W}}$.

A sequential program S is a function from a set of the form $\{0, 1, \dots, N\}$ (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 $\mathsf{Tid} \subseteq \{\mathtt{T}_1, \mathtt{T}_2, \dots\}$ 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 $\mathtt{T}_1, \mathtt{T}_2, \dots$ 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 Σ is a triple $\langle Q, Q_0, T\rangle$, 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 $A.\mathsf{Q}$, $A.\mathsf{Q}_0$ and $A.\mathsf{T}$ the components of an LTS A; write $\xrightarrow{\sigma}_A$ for the relation $\{\langle q, q'\rangle \mid \langle q, \sigma, q'\rangle \in A.\mathsf{T}\}$ and $\rightarrow_A$ for $\bigcup_{\sigma\in\Sigma} \xrightarrow{\sigma}_A$. A state $q \in A.\mathsf{Q}$ is *reachable* in A if $q_0 \rightarrow^*_A q$ for some $q_0 \in A.\mathsf{Q}_0$. A sequence $\sigma_1, \dots, \sigma_n$ is a *trace* of A if $q_0 \xrightarrow{\sigma_1}_A \cdots \xrightarrow{\sigma_n}_A q$ for some $q_0 \in A.\mathsf{Q}_0$ and $q \in A.\mathsf{Q}$. The set of *predecessors* of a set $S \subseteq A.\mathsf{Q}$ *w.r.t. a symbol* $\sigma \in \Sigma$, denoted by $\mathsf{pred}^{\sigma}_A(S)$, is given by $\{q \in A.\mathsf{Q} \mid \exists q' \in S.\ q \xrightarrow{\sigma}_A q'\}$. We define $\mathsf{pred}_A(S) \triangleq \bigcup_{\sigma\in\Sigma} \mathsf{pred}^{\sigma}_A(S)$.

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

Definition 2.1. A *label* is either $\mathtt{R}(x, v_{\mathtt{R}})$ (*read label*), $\mathtt{W}(x, v_{\mathtt{W}})$ (*write label*) or $\mathtt{RMW}(x, v_{\mathtt{R}}, v_{\mathtt{W}})$ (*read-modify-write label*), where $x \in \mathsf{Loc}$ and $v_{\mathtt{R}}, v_{\mathtt{W}} \in \mathsf{Val}$. We denote by Lab the set of all labels. The functions $\mathtt{typ}$, $\mathtt{loc}$, $\mathtt{val}_{\mathtt{R}}$, and $\mathtt{val}_{\mathtt{W}}$ return (when applicable) the type ($\mathtt{R}/\mathtt{W}/\mathtt{RMW}$), location, read value and written value of a given label l.

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

In turn, a concurrent program P is identified with an LTS over $\mathsf{Tid} \times (\mathsf{Lab} \cup \{\varepsilon\})$. Its states are functions, often denoted by $\overline{p}$, assigning a state in $P(\tau).\mathsf{Q}$ to every $\tau \in \mathsf{Tid}$; its initial states set is $\{\overline{p} \mid \forall\tau.\ \overline{p}(\tau) \in P(\tau).\mathsf{Q}_0\}$; and its transitions are

$v \in \mathsf{Val} \subseteq \mathbb{N}$	values
$x, y, z \in \mathsf{Loc} \subseteq \{\mathtt{x}, \mathtt{y}, \dots\}$	locations
$r \in \mathsf{Reg} \subseteq \{\mathtt{a}, \mathtt{b}, \dots\}$	registers
$\tau, \pi, \eta \in \mathsf{Tid} \subseteq \{\mathtt{T}_1, \mathtt{T}_2, \dots\}$	thread identifiers
$S \in \mathsf{SProg} \triangleq \{0, 1, \dots, N\} \to \mathsf{Inst}$	sequential programs
$P : \mathsf{Tid} \to \mathsf{SProg}$	(concurrent) programs

$$\begin{array}{rl} & e ::= r \mid v \mid e + e \mid e = e \mid e \neq e \mid \dots \\ \mathsf{Inst} \ni \mathit{inst} ::= & r := e \mid \mathtt{if}\ e\ \mathtt{goto}\ n \mid x := e \mid r := x \mid \\ & r := \mathtt{FADD}(x, e) \mid r := \mathtt{XCHG}(x, e) \mid r := \mathtt{CAS}(x, e, e) \end{array}$$

Figure 1. Domains, metavariables and programming language syntax.

$$\frac{\begin{array}{c} S(pc) = r := e \\ \phi' = \phi[r \mapsto \phi(e)] \end{array}}{\langle pc, \phi \rangle \xrightarrow{\varepsilon} \langle pc + 1, \phi' \rangle} \qquad \frac{\begin{array}{c} S(pc) = \mathtt{if}\ e\ \mathtt{goto}\ n \\ \phi(e) \neq 0 \end{array}}{\langle pc, \phi \rangle \xrightarrow{\varepsilon} \langle n, \phi \rangle} \qquad \frac{\begin{array}{c} S(pc) = \mathtt{if}\ e\ \mathtt{goto}\ n \\ \phi(e) = 0 \end{array}}{\langle pc, \phi \rangle \xrightarrow{\varepsilon} \langle pc + 1, \phi \rangle} \qquad \frac{\begin{array}{c} S(pc) = x := e \\ l = \mathtt{W}(x, \phi(e)) \end{array}}{\langle pc, \phi \rangle \xrightarrow{l} \langle pc + 1, \phi \rangle} \qquad \frac{\begin{array}{c} S(pc) = r := x \\ l = \mathtt{R}(x, v) \quad \phi' = \phi[r \mapsto v] \end{array}}{\langle pc, \phi \rangle \xrightarrow{l} \langle pc + 1, \phi' \rangle}$$

$$\frac{\begin{array}{c} S(pc) = r := \mathtt{FADD}(x, e) \\ l = \mathtt{RMW}(x, v, v + \phi(e)) \\ \phi' = \phi[r \mapsto v] \end{array}}{\langle pc, \phi \rangle \xrightarrow{l} \langle pc + 1, \phi' \rangle} \qquad \frac{\begin{array}{c} S(pc) = r := \mathtt{XCHG}(x, e) \\ l = \mathtt{RMW}(x, v, \phi(e)) \\ \phi' = \phi[r \mapsto v] \end{array}}{\langle pc, \phi \rangle \xrightarrow{l} \langle pc + 1, \phi' \rangle} \qquad \frac{\begin{array}{c} S(pc) = r := \mathtt{CAS}(x, e_{\mathtt{R}}, e_{\mathtt{W}}) \\ l = \mathtt{RMW}(x, \phi(e_{\mathtt{R}}), \phi(e_{\mathtt{W}})) \\ \phi' = \phi[r \mapsto \phi(e_{\mathtt{R}})] \end{array}}{\langle pc, \phi \rangle \xrightarrow{l} \langle pc + 1, \phi' \rangle} \qquad \frac{\begin{array}{c} S(pc) = r := \mathtt{CAS}(x, e_{\mathtt{R}}, e_{\mathtt{W}}) \\ l = \mathtt{R}(x, v) \qquad v \neq \phi(e_{\mathtt{R}}) \\ \phi' = \phi[r \mapsto v] \end{array}}{\langle pc, \phi \rangle \xrightarrow{l} \langle pc + 1, \phi' \rangle}$$

Figure 2. Transitions of LTS induced by a sequential program $S \in \mathsf{SProg}$.

"interleaved transitions" of P's components, given by:

$$\frac{l \in \mathsf{Lab} \qquad \overline{p}(\tau) \xrightarrow{l}_{P(\tau)} s'}{\overline{p} \xrightarrow{\tau, l} \overline{p}[\tau \mapsto s']} \qquad \frac{\overline{p}(\tau) \xrightarrow{\varepsilon}_{P(\tau)} s'}{\overline{p} \xrightarrow{\tau, \varepsilon} \overline{p}[\tau \mapsto s']}$$

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

Our representation of events induces a partial order $<$ on them: events of the same thread are ordered according to their serial numbers (i.e., $\langle \tau_1, n_1, l_1 \rangle < \langle \tau_2, n_2, l_2 \rangle$ iff $\tau_1 = \tau_2$ 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 $\langle w, r \rangle \in \mathit{rf}$, then $w \in E \cap \mathsf{W}$, $r \in E \cap \mathsf{R}$, $\mathtt{loc}(w) = \mathtt{loc}(r)$ and $\mathtt{val}_{\mathtt{W}}(w) = \mathtt{val}_{\mathtt{R}}(r)$.
- If $\langle w_1, r \rangle, \langle w_2, r \rangle \in \mathit{rf}$, then $w_1 = w_2$ (that is, $\mathit{rf}^{-1} = \{\langle r, w \rangle \mid \langle w, r \rangle \in \mathit{rf}\}$ is functional).
- $\forall r \in E \cap \mathsf{R}.\ \exists w.\ \langle w, r \rangle \in \mathit{rf}$ (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 is a disjoint union of relations $\{\mathit{mo}_x\}_{x \in \mathsf{Loc}}$ where each mo_x is a strict total order on $E \cap \mathsf{W}_x$.

Definition 2.5. An *execution graph* is a triple $G = \langle E, \mathit{rf}, \mathit{mo} \rangle$ 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 EGraph the set of all execution graphs. The components of G are denoted by $G.\mathtt{E}$, $G.\mathtt{rf}$ and $G.\mathtt{mo}$, and $G.\mathtt{po}$ denotes the restriction of $<$ to $G.\mathtt{E}$ (i.e., $G.\mathtt{po} \triangleq \{\langle e_1, e_2 \rangle \in E \times E \mid e_1 < e_2\}$). For a set $E \subseteq \mathsf{E}$, we write $G.E$ for $G.\mathtt{E} \cap E$ (e.g., $G.\mathsf{W}_x = G.\mathtt{E} \cap \mathsf{W}_x$).

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 $\tau \in \mathsf{Tid}$ and label $l \in \mathsf{Lab}$, $\mathsf{NextEvent}(E, \tau, l)$ denotes the event given by $\langle \tau, 1 + \max(\{n \in \mathbb{N} \mid \exists l' \in \mathsf{Lab}.\ \langle \tau, n, l' \rangle \in E\}), l \rangle$.

Definition 2.7. An execution graph G is *generated by a program P with final state $\overline{p}$* if $\langle \overline{p}_0, G_\emptyset \rangle \to^* \langle \overline{p}, G \rangle$ for some $\overline{p}_0 \in P.\mathsf{Q}_0$, where $G_\emptyset$ denotes the empty execution graph (given by $G_\emptyset \triangleq \langle \emptyset, \emptyset, \emptyset \rangle$) and $\to$ is defined by:

$$\frac{\begin{array}{c} \overline{p} \xrightarrow{\tau, l}_P \overline{p}' \qquad E' = E \cup \{\mathsf{NextEvent}(E, \tau, l)\} \\ \mathit{rf} \subseteq \mathit{rf}' \qquad \mathit{mo} \subseteq \mathit{mo}' \end{array}}{\langle \overline{p}, \langle E, \mathit{rf}, \mathit{mo} \rangle \rangle \to \langle \overline{p}', \langle E', \mathit{rf}', \mathit{mo}' \rangle \rangle} \qquad \frac{\overline{p} \xrightarrow{\tau, \varepsilon}_P \overline{p}'}{\langle \overline{p}, G \rangle \to \langle \overline{p}', G \rangle}$$

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).

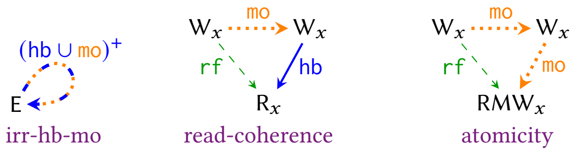
Notation 3.1 (Relations)**.** Given a relation R, $dom(R)$ denotes its domain; $R^?$ and R^+ denote its reflexive and transitive closures; and R^{-1} denotes its inverse. The (left) composition of relations R_1, R_2 is denoted by $R_1\,;R_2$. We denote by $[A]$ the identity relation on a set A, and so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

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

$$G.\mathtt{hb} \triangleq (G.\mathtt{po} \cup G.\mathtt{rf})^+$$

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 reads synchronize with the writes they read from ($\mathtt{rf} \subseteq \mathtt{hb}$), 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:



irr-hb-mo. This constraint requires that the modification order mo "agrees" with the causality order:

$$(G.\mathtt{hb} \cup G.\mathtt{mo})^+ \text{ is irreflexive} \qquad \text{(irr-hb-mo)}$$

In particular, it implies that $G.\mathtt{hb}$ 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].

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 the modification order mo to interpret one write being "later" than another, the precise condition requires that:

$$G.\mathtt{mo}\,;G.\mathtt{hb}\,;G.\mathtt{rf}^{-1} \text{ is irreflexive} \qquad \text{(read-coherence)}$$

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 $\langle w_1, w_2\rangle \in \mathtt{mo}$, $\langle w_2, r\rangle \in \mathtt{hb}$ and $\langle r, w_1\rangle \in \mathtt{rf}^{-1}$.

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:

$$G.\mathtt{mo}\,;G.\mathtt{mo}\,;G.\mathtt{rf}^{-1} \text{ is irreflexive} \qquad \text{(atomicity)}$$

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:
Input: a program P and a "bad" state $\overline{p} \in P.\mathtt{Q}$.
Question: is $\overline{p}$ 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. [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:

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 `lwsync` 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 [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 $vis = \mathtt{hb}$ and taking ar to be some total order extending $\mathtt{hb} \cup \mathtt{mo}$. 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 mo = $\bigcup_{x \in \mathsf{Loc}} [\mathsf{W}_x] \,;\, ar \,;\, [\mathsf{W}_x]$. 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 $r := \mathtt{XCHG}(x, e)$), 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

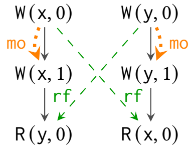
We list some well-known litmus tests to demonstrate SRA (some of which are revisited in the sequel). To simplify the presentation, instead of referring to reachable program states, we consider possible *program outcomes* assigning final values to (some) registers. An outcome $O : \mathsf{Reg} \rightharpoonup \mathsf{Val}$ 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 $G_\emptyset$ 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.

```
x := 0      || y := 0
x := 1      || y := 1
a := y //0  || b := x //0
```

✓ SRA



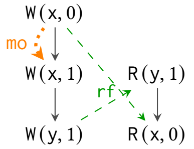
(SB)

In its execution graph the rf-edges are forced because of the read values, whereas the mo-edges are forced due to 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:

```
x := 0 || a := y //1
x := 1 || b := x //0
y := 1 ||
```

✗ SRA



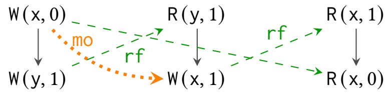
(MP)

An execution graph for this outcome must have rf and mo-edges as depicted above. However, we have mo 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 (Transitive message passing)**.** po and rf edges equally contribute to hb in causal consistency. Hence, as in Ex. 3.6, the following outcome is disallowed by SRA.

```
x := 0 || a := y //1 || b := x //1
y := 1 || x := 1     || c := x //0
```

✗ SRA

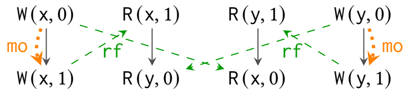


(MP-trans)

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).

```
x := 0 || a := x //1 || c := y //1 || y := 0
x := 1 || b := y //0 || d := x //0 || y := 1
```

✓ SRA

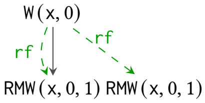


(IRIW)

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

```
x := 0              ||
a :=                || b :=
CAS(x, 0, 1) //0    || CAS(x, 0, 1) //0
```

✗ SRA

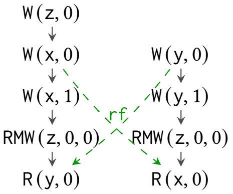


(2RMW)

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.

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

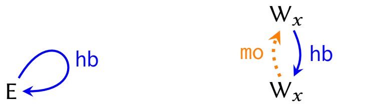
```
z := 0
x := 0              ||  y := 0
x := 1              ||  y := 1
a := FADD(z, 0)     ||  a := FADD(z, 0)
b := y // 0         ||  c := x // 0
          ✗SRA
```



(SBF)

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:



irr-hb write-coherence

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

$$G.\mathtt{hb} \text{ is irreflexive} \qquad \text{(irr-hb)}$$

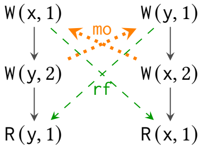
Second, instead of a *global* agreement between mo and hb, RA only requires a *local* agreement:

$$G.\mathtt{mo} \,;\, G.\mathtt{hb} \text{ is irreflexive} \qquad \text{(write-coherence)}$$

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

Example 3.11. Cycles in hb∪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 $[\mathsf{W}_x] \,;\, \mathtt{mo} \,;\, [\mathsf{W}_x]$ and $[\mathsf{W}_y] \,;\, \mathtt{mo} \,;\, [\mathsf{W}_y]$ also when $x \neq y$. The following example (adapted from [50]) demonstrates the difference:

```
x := 1         ||  y := 1
y := 2         ||  x := 2
a := y // 1    ||  a := x // 1
  ✓ RA       ✗ SRA
```



(2+2W)

An execution graph for this outcome must have rf and mo-edges as depicted above (to satisfy read-coherence), and it contains a (hb ∪ 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 $w_1, w_2 \in G.\mathsf{W}$ with $\mathtt{loc}(w_1) = \mathtt{loc}(w_2)$, we have $w_1 = w_2$, $\langle w_1, w_2 \rangle \in G.\mathtt{hb}$ or $\langle w_2, w_1 \rangle \in G.\mathtt{hb}$. 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 $\mathcal{G}$ 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 $\mathcal{G}$ is empty.

Suppose otherwise and let G be a minimal element in $\mathcal{G}$, in the sense that every proper $G.\mathtt{hb}$-prefix of G is not in $\mathcal{G}$. (A proper $G.\mathtt{hb}$-prefix of G is an execution graph of the form $\langle E_p, [E_p] \,;\, G.\mathtt{rf} \,;\, [E_p], [E_p] \,;\, G.\mathtt{mo} \,;\, [E_p] \rangle$ where $E_p \subsetneq G.\mathsf{E}$ and $dom(G.\mathtt{hb} \,;\, [E_p]) \subseteq E_p$.) Since the empty execution graph $G_\emptyset$ is trivially SRA-consistent, G cannot be empty. Let e be a $G.\mathtt{hb}$-maximal event in $G.\mathsf{E}$, and let $E' = G.\mathsf{E} \setminus \{e\}$. The minimality of G ensures that $G' = \langle E', [E'] \,;\, G.\mathtt{rf} \,;\, [E'], [E'] \,;\, G.\mathtt{mo} \,;\, [E'] \rangle$ (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 $G'.\mathtt{mo} \subseteq G'.\mathtt{hb} \subseteq G.\mathtt{hb}$.

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 $w \in E'$ such that $\langle e, w \rangle \in G.\mathtt{mo}$ and $\langle w, e \rangle \in (G.\mathtt{hb} \cup G'.\mathtt{mo})^+$. Since $G'.\mathtt{mo} \subseteq G.\mathtt{hb}$, it follows that $\langle e, e \rangle \in G.\mathtt{mo} \,;\, G.\mathtt{hb}$. 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 $(\mathsf{Tid} \times \mathsf{Lab}) \cup \{\varepsilon\}$.

$$\text{WRITE}\quad \frac{\begin{array}{c} G = \langle E, \mathit{rf}, \mathit{mo} \rangle \\ e = \mathsf{NextEvent}(G.\mathsf{E}, \tau, \mathtt{W}(x, v_{\mathtt{W}})) \\ G' = \langle E \cup \{e\}, \mathit{rf}, \mathit{mo} \cup (G.\mathsf{W}_x \times \{e\}) \rangle \end{array}}{G \xrightarrow{\tau, \mathtt{W}(x, v_{\mathtt{W}})}_{\mathsf{opSRA}} G'}$$

$$\text{READ}\quad \frac{\begin{array}{c} G = \langle E, \mathit{rf}, \mathit{mo} \rangle \\ e = \mathsf{NextEvent}(G.\mathsf{E}, \tau, \mathsf{R}(x, v_{\mathsf{R}})) \\ G' = \langle E \cup \{e\}, \mathit{rf} \cup \{\langle w, e \rangle\}, \mathit{mo} \rangle \\ w \in G.\mathsf{W}_x \qquad \mathtt{val}_{\mathtt{W}}(w) = v_{\mathsf{R}} \\ w \notin \mathit{dom}(\mathit{mo}\,;G.\mathsf{hb}^?\,;[\mathsf{E}^\tau]) \end{array}}{G \xrightarrow{\tau, \mathsf{R}(x, v_{\mathsf{R}})}_{\mathsf{opSRA}} G'}$$

$$\text{RMW}\quad \frac{\begin{array}{c} G = \langle E, \mathit{rf}, \mathit{mo} \rangle \\ e = \mathsf{NextEvent}(G.\mathsf{E}, \tau, \mathtt{RMW}(x, v_{\mathsf{R}}, v_{\mathtt{W}})) \\ G' = \langle E \cup \{e\}, \mathit{rf} \cup \{\langle w, e \rangle\}, \mathit{mo} \cup (G.\mathsf{W}_x \times \{e\}) \rangle \\ w \in G.\mathsf{W}_x \qquad \mathtt{val}_{\mathtt{W}}(w) = v_{\mathsf{R}} \\ w \notin \mathit{dom}(\mathit{mo}) \end{array}}{G \xrightarrow{\tau, \mathtt{RMW}(x, v_{\mathsf{R}}, v_{\mathtt{W}})}_{\mathsf{opSRA}} G'}$$

Figure 3. Transitions of opSRA.

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.

Example 4.2. The most well-known memory system is the one of sequential consistency, denoted here by SC. This memory system simply tracks the most recent value written to each location (or $\bot$ for uninitialized locations). Formally, it is defined by $\mathsf{SC.Q} \triangleq \mathsf{Loc} \to (\mathsf{Val} \cup \{\bot\})$, $\mathsf{SC.Q}_0 \triangleq \{\lambda x \in \mathsf{Loc}.\,\bot\}$ and $\to_{\mathsf{SC}}$ is given by:

$$\frac{\mu' = \mu[x \mapsto v_{\mathtt{W}}]}{\mu \xrightarrow{\tau, \mathtt{W}(x, v_{\mathtt{W}})}_{\mathsf{SC}} \mu'} \qquad \frac{\mu(x) = v_{\mathsf{R}}}{\mu \xrightarrow{\tau, \mathsf{R}(x, v_{\mathsf{R}})}_{\mathsf{SC}} \mu} \qquad \frac{\begin{array}{c} \mu(x) = v_{\mathsf{R}} \\ \mu' = \mu[x \mapsto v_{\mathtt{W}}] \end{array}}{\mu \xrightarrow{\tau, \mathtt{RMW}(x, v_{\mathsf{R}}, v_{\mathtt{W}})}_{\mathsf{SC}} \mu'}$$

Note that SC is oblivious to the thread that takes the action ($\mu \xrightarrow{\tau, l}_{\mathsf{SC}} \mu'$ iff $\mu \xrightarrow{\pi, l}_{\mathsf{SC}} \mu'$), 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 $(\mathsf{Tid} \times (\mathsf{Lab} \cup \{\varepsilon\})) \cup \{\varepsilon\}$ whose set of states is $P.\mathsf{Q} \times M.\mathsf{Q}$; its initial states set is $P.\mathsf{Q}_0 \times M.\mathsf{Q}_0$; and its transitions are "synchronized transitions" of P and M, given by:

$$\frac{\begin{array}{c} l \in \mathsf{Lab} \quad \overline{p} \xrightarrow{\tau, l}_P \overline{p}' \\ m \xrightarrow{\tau, l}_M m' \end{array}}{\langle \overline{p}, m \rangle \xrightarrow{\tau, l}_{P_M} \langle \overline{p}', m' \rangle} \qquad \frac{\overline{p} \xrightarrow{\tau, \varepsilon}_P \overline{p}'}{\langle \overline{p}, m \rangle \xrightarrow{\tau, \varepsilon}_{P_M} \langle \overline{p}', m \rangle} \qquad \frac{m \xrightarrow{\varepsilon}_M m'}{\langle \overline{p}, m \rangle \xrightarrow{\varepsilon}_{P_M} \langle \overline{p}, m' \rangle}$$

Next, we present the memory system opSRA that is equivalent to 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 opSRA for the SB example.

The states of opSRA are execution graphs capturing (partially ordered) histories of executed actions ($\mathsf{opSRA.Q} \triangleq \mathsf{EGraph}$); the (only) initial state is the empty execution graph $G_\emptyset$ ($\mathsf{opSRA.Q}_0 \triangleq \{G_\emptyset\}$); and the transitions are given in Fig. 3. A WRITE step by thread τ adds a corresponding fresh write event e to the graph placed after all events of thread τ and extends mo to order e *after* all existing writes to the same location. A READ step by thread τ 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 ($w \in G.\mathsf{W}_x$), writing the value being read ($\mathtt{val}_{\mathtt{W}}(w) = v_{\mathsf{R}}$), and the thread executing the read is not aware of an mo-later write to the same location ($w \notin \mathit{dom}(\mathit{mo}\,;\mathsf{hb}^?\,;[\mathsf{E}^\tau])$). An RMW step combines a READ and a WRITE, but it is enforced to pick the mo-*maximal* write to the relevant location in the current graph as the reads-from source of the freshly added RMW.

This semantics exploits the fact that $\mathsf{hb} \cup \mathit{mo}$ is acyclic in SRA-consistent execution graphs (as per 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 $\mathsf{hb} \cup \mathit{mo}$, which guarantees that writes are executed following their mo-order. In turn, since RMWs should read from their immediate mo-predecessor, we require that RMWs read from the current mo-maximal write.

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

Definition 4.4. A state $\overline{p}$ of a program P is *reachable* under a memory system M if $\langle \overline{p}, m \rangle$ is reachable in P_M for some $m \in M.\mathsf{Q}$.

Theorem 4.5. *A state $\overline{p}$ 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 opSRA by following any total order extending $G.\mathsf{hb} \cup G.\mathit{mo}$. 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 $\langle \overline{p}, G \rangle$ is reachable in opSRA, then G is an SRA-consistent execution graph that is generated by P with final state $\overline{p}$. □

Remark 2. Following [27], our formulation of 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 $\mathtt{loc}(o)$, a value $\mathtt{val}(o)$ 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 $o_1 \cdot o_2$ allows the thread to read $\mathtt{val}(o_1)$ from location $\mathtt{loc}(o_1)$ and then $\mathtt{val}(o_2)$ from location $\mathtt{loc}(o_2)$. 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 (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 $\mathtt{val}(o) = v$ and $\mathtt{loc}(o) = x$ 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 $\mathtt{loc}(o) = x$ 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 $\mathtt{loc}(o) = x$ and $\mathtt{val}(o) = v$ 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^v_x 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 $\mathtt{T}_1$ or $\mathtt{T}_4$. Suppose, w.l.o.g., that $\mathtt{T}_1$ begins. Since $\mathtt{T}_3$ reads 0 from x, a read option $o^0_{\mathtt{x}}$ should be added in the lists of $\mathtt{T}_3$. Now, before reading 0 from x, $\mathtt{T}_3$ has to read 1 from y. Hence, when $\mathtt{T}_4$ writes 1 to y, a read option $o^1_{\mathtt{y}}$ should be placed *before* $o^0_{\mathtt{x}}$ in the lists of $\mathtt{T}_3$.

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, $\mathtt{T}_1$ writes 0 to x and adds a corresponding option $o^0_{\mathtt{x}}$ to the (initially empty) list of $\mathtt{T}_2$, and then writes 1 to x without adding any option to any list (no thread reads 1 from x in this program outcome). Then, $\mathtt{T}_1$ further writes 1 to y and adds a corresponding option $o^1_{\mathtt{y}}$ in the list of $\mathtt{T}_1$ placed *before* $o^0_{\mathtt{x}}$. Finally, $\mathtt{T}_2$ may run: read 1 from y (consuming $o^1_{\mathtt{y}}$) and then 0 from x (consuming $o^0_{\mathtt{x}}$).

The restriction we impose on the positions of the added read options stems from the following key observation:[2]

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 $o^1_{\mathtt{y}}$ *cannot* be placed before $o^0_{\mathtt{x}}$, because $\mathtt{T}_1$ cannot have $o^0_{\mathtt{x}}$ 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      || y := 0      || z := 0      ||             ||             ||
x := 1      || y := 1      || z := 1      || d1 := x //1 || e1 := y //1 || f1 := z //1
a1 := z //1 || b1 := x //1 || c1 := y //1 || d2 := y //1 || e2 := z //1 || f2 := x //1
a2 := y //0 || b2 := z //0 || c2 := x //0 || d3 := z //0 || e3 := x //0 || f3 := y //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, $\mathtt{T}_3$ has to read 1 from y and 0 from x. Hence, its read-option list must include $o^1_{\mathtt{y}}$ and $o^0_{\mathtt{x}}$ in this order. In addition, a read option $o^1_{\mathtt{z}}$ should be placed in $\mathtt{T}_6$'s list before $o^1_{\mathtt{x}} \cdot o^0_{\mathtt{y}}$. The justification for it requires $o^1_{\mathtt{x}} \cdot o^0_{\mathtt{y}}$ to be a subsequence of $\mathtt{T}_3$'s list. This implies that $\mathtt{T}_3$'s list should contain some interleaving of $o^1_{\mathtt{y}} \cdot o^0_{\mathtt{x}}$ and $o^1_{\mathtt{x}} \cdot o^0_{\mathtt{y}}$. But, no such interleaving is a possible future for $\mathtt{T}_3$ (and thus cannot be generated by loSRA): reading $o^1_{\mathtt{y}}$ does not allow $\mathtt{T}_3$ to read

[2] A weaker observation, which only considers single reads, was essential for the soundness of OGRA—an Owicki Gries logic for RA introduced in [32].

o^0_{y} later; and reading o^1_{x} does not allow T_3 to read o^0_{x} later. By allowing more than one read-option list per thread, we can have $o^1_{\mathsf{y}} \cdot o^0_{\mathsf{x}}$ and $o^1_{\mathsf{x}} \cdot o^0_{\mathsf{y}}$ in two separate lists in the potential of T_3—both are possible continuations for it after z := 1. Then, after executing z := 1, T_3 may "lose" the justifying list $o^1_{\mathsf{x}} \cdot o^0_{\mathsf{y}}$, and choose to continue with $o^1_{\mathsf{y}} \cdot o^0_{\mathsf{x}}$ 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:

$$\begin{array}{l||l||l||l} \mathsf{x} := 0 & \mathsf{y} := 0 & \mathsf{a} := \mathsf{z} \;/\!/\, 1 & \\ \mathsf{x} := 1 & \mathsf{y} := 1 & \mathsf{w} := 1 & \begin{array}{l} \mathsf{c} := \mathsf{w} \;/\!/\, 1 \\ \mathsf{d} := \mathsf{y} \;/\!/\, 0 \end{array} \\ \mathsf{z} := 1 & \mathsf{z} := 1 & \mathsf{b} := \mathsf{x} \;/\!/\, 0 & \end{array} \qquad ✗\,\text{SRA} \qquad \text{(2MP)}$$

Its annotated outcome is disallowed under SRA. Indeed, since T_3 reads x = 0 after z = 1, the read of z must read from the write of T_2. But then, T_4, after reading w = 1 (from T_3) cannot read y = 0. However, the semantics described so far allows this outcome as in the following snippet:

$$\{\epsilon\} \,||\, \{\epsilon\} \,||\, \{\epsilon\} \,||\, \{\epsilon\} \xrightarrow[\mathsf{W(x,0)}]{\mathsf{T}_1} \xrightarrow[\mathsf{W(x,1)}]{\mathsf{T}_1} \xrightarrow[\mathsf{W(y,0)}]{\mathsf{T}_2} \xrightarrow[\mathsf{W(y,1)}]{\mathsf{T}_2} \xrightarrow[\mathsf{W(z,1)}]{\mathsf{T}_1}$$
$$\{o^0_{\mathsf{y}}\} \,||\, \{o^0_{\mathsf{x}}\} \,||\, \{o^0_{\mathsf{x}}, o^1_{\mathsf{z}}o^0_{\mathsf{y}}\} \,||\, \{o^0_{\mathsf{y}}\} \xrightarrow[\mathsf{W(z,1)}]{\mathsf{T}_2} \{o^0_{\mathsf{y}}\} \,||\, \{o^0_{\mathsf{x}}\} \,||\, \{o^1_{\mathsf{z}}o^0_{\mathsf{x}}, o^1_{\mathsf{z}}o^0_{\mathsf{y}}\} \,||\, \{o^0_{\mathsf{y}}\}$$
$$\xrightarrow[\mathsf{R(z,1)}]{\mathsf{T}_3} \{o^0_{\mathsf{y}}\} \,||\, \{o^0_{\mathsf{x}}\} \,||\, \{o^0_{\mathsf{x}}, o^0_{\mathsf{y}}\} \,||\, \{o^0_{\mathsf{y}}\} \xrightarrow[\mathsf{W(w,1)}]{\mathsf{T}_3} \{o^0_{\mathsf{y}}\} \,||\, \{o^0_{\mathsf{x}}\} \,||\, \{o^0_{\mathsf{x}}, o^0_{\mathsf{y}}\} \,||\, \{o^1_{\mathsf{w}}o^0_{\mathsf{y}}\} \ldots$$

What went wrong? The problem arises when T_3 reads 1 from z. At this point it has two possible futures, $o^1_{\mathsf{z}}o^0_{\mathsf{x}}$ and $o^1_{\mathsf{z}}o^0_{\mathsf{y}}$. Since read options, consisting of location and value, do not uniquely identify writes, it may read 1 from z, and remain with both o^0_{x} and o^0_{y}. Now, it uses one of these options to justify the position of o^1_{w} in the list of T_4, and the other for its own read. However, in a single run of opSRA, when reading 1 from z, T_3 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 τ to read v from x, a read option o with $\mathtt{val}(o) = v$ and $\mathtt{loc}(o) = x$ and some *unique* writing thread identifier must be the first in every read-option list of τ. In this example, the two o^1_{z} options will have different thread identifiers, which forces T_3 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 mo-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 mo-maximal write. Accordingly, an RMW step may only consume read options marked as RMW. Since WRITE steps to x replace the mo-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 ϵ 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 σ appears in s, that is if $s(k) = \sigma$ for some $1 \leq k \leq |s|$. We use "$\cdot$" 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 $o = \langle \tau, x, v, u \rangle$, where $\tau \in \mathsf{Tid}$, $x \in \mathsf{Loc}$, $v \in \mathsf{Val}$ and $u \in \{\mathsf{R}, \mathsf{RMW}\}$. The functions $\mathtt{tid}$, $\mathtt{loc}$, $\mathtt{val}$ and $\mathtt{rmw}$ return the thread identifier (τ), 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: $\mathcal{B} \sqsubseteq \mathcal{B}'$ if $\mathcal{B}(\tau) \sqsubseteq \mathcal{B}'(\tau)$ for every $\tau \in \mathsf{Tid}$.

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: loSRA.Q is the set of functions $\mathcal{B}$ assigning a potential to every $\tau \in \mathsf{Tid}$; $\text{loSRA.Q}_0 = \{\lambda\tau \in \mathsf{Tid}.\ \{\epsilon\}\}$;[3] and the transitions are given in Fig. 4.

[3] To achieve implicit initialization of all locations to 0, one should take loSRA.Q_0 to consist of all functions assigning to each thread sequences consisting of read options of the form $\langle \mathsf{T}_0, x, 0, u \rangle$ where T_0 is a distinguished thread identifier that is not used in programs (corresponds to the initializing thread, see Remark 1).

$$\textsc{write}\ \frac{\begin{array}{l}\forall \pi \in \mathsf{Tid}, L' \in \mathcal{B}'(\pi).\ \exists n \geq 0, u_1, \dots, u_n, L_0, \dots, L_n.\\ \quad L' = L_0 \cdot \langle \tau, x, v_{\mathtt{W}}, u_1 \rangle \cdot L_1 \cdot \dots \cdot \langle \tau, x, v_{\mathtt{W}}, u_n \rangle \cdot L_n\\ \wedge\ L_0 \cdot \dots \cdot L_n \in \mathcal{B}(\pi)\ \wedge\ L_1 \cdot \dots \cdot L_n \in \mathcal{B}(\tau)\\ \wedge\ (\pi = \tau \implies \forall o \in L_0 \cdot \dots \cdot L_n.\ \mathtt{loc}(o) \neq x)\\ \wedge\ \forall o \in L_0 \cdot \dots \cdot L_n.\ \mathtt{loc}(o) = x \implies \mathtt{rmw}(o) = \mathtt{R}\end{array}}{\mathcal{B} \xrightarrow{\tau, \mathtt{W}(x, v_{\mathtt{W}})}_{\mathsf{loSRA}} \mathcal{B}'}
\qquad
\textsc{read}\ \frac{\begin{array}{c}\mathtt{loc}(o) = x\\ \mathtt{val}(o) = v_{\mathtt{R}}\\ \mathcal{B} = \mathcal{B}'[\tau \mapsto o \cdot \mathcal{B}'(\tau)]\end{array}}{\mathcal{B} \xrightarrow{\tau, \mathtt{R}(x, v_{\mathtt{R}})}_{\mathsf{loSRA}} \mathcal{B}'}$$

$$\textsc{rmw}\ \frac{\begin{array}{c}\mathtt{loc}(o) = x \qquad \mathtt{val}(o) = v_{\mathtt{R}}\\ \mathtt{rmw}(o) = \mathtt{RMW}\\ \mathcal{B} = \mathcal{B}_{\mathrm{mid}}[\tau \mapsto o \cdot \mathcal{B}_{\mathrm{mid}}(\tau)]\\ \mathcal{B}_{\mathrm{mid}} \xrightarrow{\tau, \mathtt{W}(x, v_{\mathtt{W}})}_{\mathsf{loSRA}} \mathcal{B}'\end{array}}{\mathcal{B} \xrightarrow{\tau, \mathtt{RMW}(x, v_{\mathtt{R}}, v_{\mathtt{W}})}_{\mathsf{loSRA}} \mathcal{B}'}
\qquad
\textsc{lower}\ \frac{\mathcal{B}' \sqsubseteq \mathcal{B}}{\mathcal{B} \xrightarrow{\varepsilon}_{\mathsf{loSRA}} \mathcal{B}'}$$

Figure 4. Transitions of loSRA.

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 ($L_1 \cdot \dots \cdot L_n \in \mathcal{B}(\tau)$). It suffices, however, for the suffix after the first added read option to be a *subsequence* of some list of the writing thread ($\{L_1 \cdot \dots \cdot L_n\} \sqsubseteq \mathcal{B}(\tau)$). 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 $L_0 \cdot \dots \cdot L_n \in \mathcal{B}(\tau)$ implies $\{L_1 \cdot \dots \cdot L_n\} \sqsubseteq \mathcal{B}(\tau)$).

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 $\mathcal{B}'(\tau)$ is non-empty, and so the set $\mathcal{B}(\tau)$ as defined in the step is non-empty. When all options are consumed, τ'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 loSRA 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 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 $\langle \pi, \mathsf{y}, 1, u \rangle$ 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 loSRA, the x := 0 by T_1 must add a read option $\langle \mathsf{T}_1, \mathsf{x}, 0, \mathtt{RMW} \rangle$ 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 loSRA.

Next, we establish the equivalence of loSRA and opSRA. To do so, we define a relation $\vee \subseteq \mathsf{loSRA}.\mathtt{Q} \times \mathsf{opSRA}.\mathtt{Q}$, formalizing the intuitive simulation discussed so far between loSRA's lists and opSRA's execution graphs. For defining $\vee$, 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 $\langle G, L \rangle$*-write-list* if $|L| = |W|$ and the following hold for every $1 \leq k \leq |W|$ with $L(k) = \langle \tau, x, v, u \rangle$:

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

A write list W is $\langle G, \tau \rangle$*-consistent* if, intuitively, the extension of G with a sequence of read events in thread τ

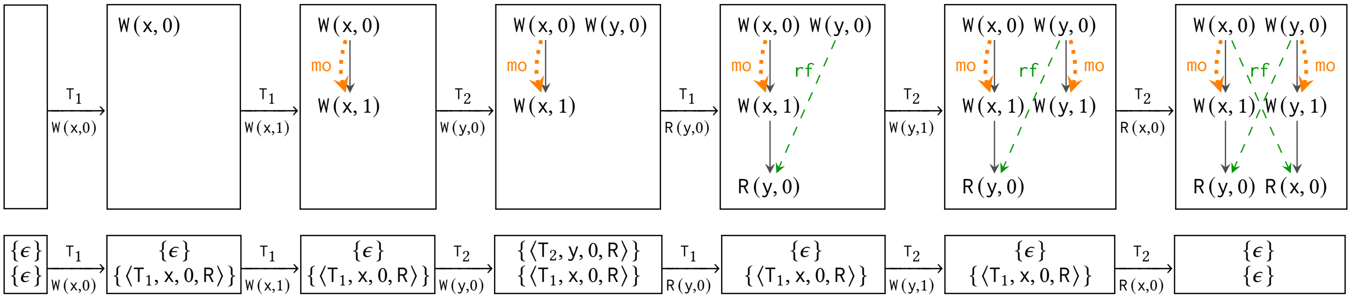


Figure 5. Illustration of runs of opSRA and loSRA for the SB program (Ex. 3.5). In opSRA's states (execution graphs), events of T_1 are on the left and of T_2 on the right. In loSRA's states (a potential for each thread), the potential of T_1 is at the top and of T_2 at the bottom. In this simple example, all option lists consist of at most one option and all potentials are singletons.

reading from the sequence of write events in W satisfies read-coherence. Thus, we ensure that thread τ 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 τ 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 $\langle G, \tau\rangle$*-consistent* if $W(k) \notin dom(G.\mathsf{mo} \,;G.\mathsf{hb}^? \,; [\mathsf{E}^\tau \cup \{W(j) \mid 1 \leq j < k\}])$ for every $1 \leq k \leq |W|$.

Now, $\curlyvee$ relates a loSRA state $\mathcal{B}$ with an execution graph G if each read-option list in $\mathcal{B}$ has an appropriate write list.

Definition 5.13. A state $\mathcal{B} \in \mathsf{loSRA.Q}$ *matches* an execution graph G, denoted by $\mathcal{B} \curlyvee G$, if for every $\tau \in \mathsf{Tid}$ and $L \in \mathcal{B}(\tau)$, there exists a $\langle G, \tau\rangle$-consistent $\langle G, L\rangle$-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. Notice that ε-transitions do not affect reachability of problem states, which only concerns the sequence of labels of the program. As loSRA employs ε-transitions (LOWER), while opSRA does not, the trace equivalence ignores ε-transitions.

Definition 5.14. Two traces are *equivalent* if their restrictions to non ε-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* loSRA *there is an equivalent trace of* opSRA.

Proof Outline. We show that $\curlyvee$ 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 (i) $\mathcal{B} \curlyvee G$, witnessed by a $\langle G, \pi\rangle$-consistent $\langle G, L\rangle$-write-list $W_{\langle \pi, L\rangle}$ for every $\pi \in \mathsf{Tid}$ and $L \in \mathcal{B}(\pi)$; and (ii) $\mathcal{B} \xrightarrow{\tau,l}_{\mathsf{loSRA}} \mathcal{B}'$. We construct G' such that $\mathcal{B}' \curlyvee G'$ and $G \xrightarrow{\tau,l}_{\mathsf{opSRA}} G'$ (as depicted on the right).

$$\begin{tikzcd}
\mathcal{B} \arrow[d, "{\tau,l}"'] & \curlyvee & G \arrow[d, "{\tau,l}"] \\
\mathcal{B}' & \curlyvee & \boxed{G'}_{\exists}
\end{tikzcd}$$

We describe here the WRITE and READ steps (the 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 po inside the writing thread τ, and last in mo among the writes to the same location. Then, $G \xrightarrow{\tau,l}_{\mathsf{opSRA}} G'$ is trivial. To show that $\mathcal{B}' \curlyvee G'$, we construct for every $\pi \in \mathsf{Tid}$ and $L' \in \mathcal{B}'(\pi)$, 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 $\mathcal{B}(\pi)$; and (iii) each old option that appears after the first new option to the mo-maximal write event among (1) its mapping in the existing write list W for the corresponding list in $\mathcal{B}(\pi)$ and (2) its mapping in the existing write list W_τ for the justifying list in $\mathcal{B}(\tau)$. Roughly speaking, picking the mo-maximal write in the third case ensures that W' is $\langle G', \pi\rangle$-consistent: In the new state, thread π might not be able to read from a write that it previously read from (since it "synchronized" with τ that may be aware of a later write) and might not be able to read from the a write that τ reads from (since π 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 loSRA in opSRA, we need to pick a write event w from which the added read event r will read-from in G'. We pick w to be the po-minimal event among the write events that the $W_{\langle \tau, L\rangle}$ 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 $W_{\langle \tau, L\rangle}$ lists ensures that $w \notin dom(G.\mathsf{mo} \,; G.\mathsf{hb}^? \,; [\mathsf{E}^\tau])$, so we can make the READ step in opSRA when reading from w. To show that $\mathcal{B}' \curlyvee G'$, for every thread $\pi \neq \tau$, we simply reuse the write list $W_{\langle \pi,L \rangle}$ that we had for G; while for thread τ itself, we shift its write lists by one (setting $W'_{\langle \tau,L \rangle} \triangleq \lambda k.\ W_{\langle \tau,L \rangle}(1+k)$), and use the po-minimality of w to establish $\langle G', \tau \rangle$-consistency. □

For the converse, we favor *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 $\curlyvee^{-1}$ constitutes a backward simulation from opSRA to loSRA.

For the main proof obligation (depicted on the right), suppose that $G \xrightarrow{\tau,l}_{\mathsf{opSRA}} G'$ and $\mathcal{B}' \curlyvee G'$, witnessed by a $\langle G', \pi \rangle$-consistent $\langle G', L' \rangle$-write-list $W'_{\langle \pi,L' \rangle}$ for every $\pi \in \mathsf{Tid}$ and $L' \in \mathcal{B}'(\pi)$. We construct a state $\mathcal{B}$ such that $\mathcal{B} \xrightarrow{\tau,l}_{\mathsf{loSRA}} \mathcal{B}'$ and $\mathcal{B} \curlyvee G$.

$$\begin{tikzcd} \boxed{\exists \mathcal{B}} \arrow[d, "\tau{,}l"'] & \curlyvee & G \arrow[d, "\tau{,}l"] \\ \mathcal{B}' & \curlyvee & G' \end{tikzcd}$$

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 $\mathcal{B}$ 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 $\mathcal{B}$ are then induced by those for $\mathcal{B}'$ 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 $\mathcal{B}$ by setting $\mathcal{B} = \mathcal{B}'[\tau \mapsto \langle \mathtt{tid}(w), \mathtt{loc}(r), \mathtt{val}_{\mathsf{R}}(r), \mathsf{R} \rangle \cdot \mathcal{B}'(\tau)]$, and $\mathcal{B} \xrightarrow{\tau,l}_{\mathsf{loSRA}} \mathcal{B}'$ follows by definition. Now, to show that $\mathcal{B} \curlyvee G$, we use the write lists of $\mathcal{B}'$ for every $\pi \neq \tau$. For $\pi = \tau$ we append w in the beginning of the write lists of $\mathcal{B}'$. □

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.

Preliminaries. A *well-quasi-ordering* (wqo) on a set S is a reflexive and transitive relation $\lesssim$ on S such that for every infinite sequence $s_1, s_2, \ldots$ of elements of S, we have $s_i \lesssim s_j$ for some $i < j$. In a context of a set S and a wqo $\lesssim$ on S, the *upward closure* of a set $U \subseteq S$, denoted by $\uparrow U$, is given by $\{s \in S \mid \exists u \in U.\ u \lesssim s\}$; a set $U \subseteq S$ is called *upward closed* if $U = \uparrow U$; and a set $B \subseteq U$ is called a *basis* of U if $U = \uparrow B$. 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 $\lesssim$ on $A.\mathsf{Q}$ that is *compatible* with A, that is: if $q_1 \rightarrow_A q_2$ and $q_1 \lesssim q_3$, then there exists $q_4 \in A.\mathsf{Q}$ such that $q_3 \rightarrow^*_A q_4$ and $q_2 \lesssim q_4$. The *coverability problem* for $\langle A, \lesssim \rangle$ asks whether an input state $q \in A.\mathsf{Q}$ is coverable, namely: is some state q' with $q \lesssim q'$ reachable in A?

Coverability is decidable (see, e.g., [7, 22]) for a WSTS $\langle A, \lesssim \rangle$ provided that $\lesssim$ is decidable and the following hold:

(i) *effective initialization*: there exists an algorithm that accepts a state $q \in A.\mathsf{Q}$ and decides whether $\uparrow\{q\} \cap A.\mathsf{Q}_0 = \emptyset$.

(ii) *effective pred-basis*: there exists an algorithm that accepts a state $q \in A.\mathsf{Q}$ and returns a finite basis of $\uparrow \mathsf{pred}_A(\uparrow\{q\})$.

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.

loSRA *as a Well-Structured Transition System.* The $\sqsubseteq$ ordering on the states of loSRA is clearly decidable and also forms a wqo. Indeed, by Higman's lemma, $\sqsubseteq$ 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 $\sqsubseteq$ to functions assigning a potential to every $\tau \in \mathsf{Tid}$ (i.e., states of loSRA) is also a wqo.

Now, let P be a program. The $\sqsubseteq$ ordering is naturally lifted to states of the concurrent system P_{loSRA} (that is, pairs $\langle \overline{p}, \mathcal{B} \rangle \in P.\mathsf{Q} \times \mathsf{loSRA.Q}$, see Def. 4.3) by defining $\langle \overline{p}, \mathcal{B} \rangle \sqsubseteq \langle \overline{p}', \mathcal{B}' \rangle$ iff $\overline{p} = \overline{p}'$ and $\mathcal{B} \sqsubseteq \mathcal{B}'$.

Lemma 6.1. *P_{loSRA} equipped with $\sqsubseteq$ is a WSTS that admits effective initialization and effective pred-basis.*

Proof. First, since $P.\mathsf{Q}$ is (by definition) finite and $\sqsubseteq$ is a wqo on $\mathsf{loSRA.Q}$, we have that $\sqsubseteq$ is a wqo of $P_{\mathsf{loSRA}}.\mathsf{Q}$.

Second, since LOWER is explicitly included in loSRA, $\sqsubseteq$ is clearly compatible with P_{loSRA}. Indeed, given $q_1 = \langle \overline{p}_1, \mathcal{B}_1 \rangle$, $q_2 = \langle \overline{p}_2, \mathcal{B}_2 \rangle$ and $q_3 = \langle \overline{p}_3, \mathcal{B}_3 \rangle$ such that $q_1 \rightarrow_{P_{\mathsf{loSRA}}} q_2$ and $q_1 \sqsubseteq q_3$ (so $\overline{p}_1 = \overline{p}_3$), for $q_4 = q_2$, we have $q_3 \rightarrow^*_{P_{\mathsf{loSRA}}} q_4$ (since $\mathcal{B}_3 \xrightarrow{\varepsilon}_{\mathsf{loSRA}} \mathcal{B}_1$ using the LOWER step) and $q_2 \sqsubseteq q_4$.

Next, P_{loSRA} trivially admits effective initialization. Indeed, the states $\langle \overline{p}, \mathcal{B} \rangle$ for which $\uparrow\{\langle \overline{p}, \mathcal{B} \rangle\} \cap P_{\mathsf{loSRA}}.\mathsf{Q}_0 \neq \emptyset$ are exactly the initial states themselves—$P.\mathsf{Q}_0 \times \{\lambda\tau.\ \{\epsilon\}\}$.

Finally, [1] establishes the effective pred-basis for P_{loSRA}. For this matter, we demonstrate how to calculate a finite basis of $\uparrow \mathsf{pred}^{\alpha}_{\mathsf{loSRA}}(\uparrow\{\mathcal{B}'\})$ for α of the form $\langle \tau, \mathtt{W}(x, v_{\mathsf{W}}) \rangle$, $\langle \tau, \mathtt{R}(x, v_{\mathsf{R}}) \rangle$, $\langle \tau, \mathtt{RMW}(x, v_{\mathsf{R}}, v_{\mathsf{W}}) \rangle$ or ε. □

It is now easy to establish the decidability of reachability under loSRA.

Theorem 6.2 (loSRA reachability)**.** *Given a program P and a state $\overline{p} \in P.\mathsf{Q}$, it is decidable to check whether $\overline{p}$ is reachable (see Def. 4.4) under the memory system* loSRA.

Proof. Since the first component (the program state) in $\sqsubseteq$-ordered pairs of P_{loSRA}'s states is equal, reachability under loSRA is reduced to coverability in $\langle P_{\mathsf{loSRA}}, \sqsubseteq\rangle$, 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 $\overline{p} \in P.\mathsf{Q}$, it is decidable to check whether $\overline{p}$ 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 $\mathtt{mo}$ separately from the execution order, while SRA allows the execution of writes following $\mathtt{hb} \cup \mathtt{mo}$. (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-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. 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, 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., [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

[1] Ori Lahav and Udi Boker. 2020. Supplementary material for this paper. https://www.cs.tau.ac.il/~orilahav/papers/pldi20full.pdf

[2] Parosh Aziz Abdulla. 2010. Well (and better) quasi-ordered transition systems. *The Bulletin of Symbolic Logic* 16, 4 (2010), 457–515. http://www.jstor.org/stable/40961367

[3] Parosh Aziz Abdulla, Jatin Arora, Mohamed Faouzi Atig, and Shankaranarayanan Krishna. 2019. Verification of programs under the release-acquire semantics. In *PLDI*. ACM, New York, NY, USA, 1117–1132. https://doi.org/10.1145/3314221.3314649

[4] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Ahmed Bouajjani, and Tuan Phong Ngo. 2018. A load-buffer semantics for total store ordering. *Logical Methods in Computer Science* Volume 14, Issue 1 (Jan. 2018). https://doi.org/10.23638/LMCS-14(1:9)2018

[5] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, and Tuan Phong Ngo. 2018. Optimal stateless model checking under the release-acquire semantics. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 135 (Oct. 2018), 29 pages. https://doi.org/10.1145/3276505

[6] Parosh Aziz Abdulla, Mohamed Faouzi Atig, and Rojin Rezvan. 2019. Parameterized verification under TSO is PSPACE-complete. *Proc. ACM Program. Lang.* 4, POPL, Article 26 (Dec. 2019), 29 pages. https://doi.org/10.1145/3371094

[7] Parosh Aziz Abdulla, Kārlis Čerāns, Bengt Jonsson, and Yih-Kuen Tsay. 2000. Algorithmic analysis of programs with well quasi-ordered domains. *Information and Computation* 160, 1 (2000), 109 – 127. https://doi.org/10.1006/inco.1999.2843

[8] Sarita V. Adve and Mark D. Hill. 1990. Weak ordering—a new definition. In *ISCA*. ACM, New York, NY, USA, 2–14. https://doi.org/10.1145/325164.325100

[9] Jade Alglave, Luc Maranget, and Michael Tautschnig. 2014. Herding cats: modelling, simulation, testing, and data mining for weak memory. *ACM Trans. Program. Lang. Syst.* 36, 2, Article 7 (July 2014), 74 pages. https://doi.org/10.1145/2627752

[10] Masoud Saeida Ardekani, Pierre Sutra, and Marc Shapiro. 2013. Non-monotonic snapshot isolation: Scalable and strong consistency for geo-replicated transactional systems. In *SRDS*. IEEE Computer Society, Washington, DC, USA, 163–172. https://doi.org/10.1109/SRDS.2013.25

[11] Mohamed Faouzi Atig, Ahmed Bouajjani, Sebastian Burckhardt, and Madanlal Musuvathi. 2010. On the verification problem for weak

memory models. In *POPL*. ACM, New York, NY, USA, 7–18. https://doi.org/10.1145/1706299.1706303

[12] Mohamed Faouzi Atig, Ahmed Bouajjani, Sebastian Burckhardt, and Madanlal Musuvathi. 2012. What's decidable about weak memory models?. In *ESOP*. Springer-Verlag, Berlin, Heidelberg, 26–46. https://doi.org/10.1007/978-3-642-28869-2_2

[13] Mark Batty, Kayvan Memarian, Kyndylan Nienhuis, Jean Pichon-Pharabod, and Peter Sewell. 2015. The problem of programming language concurrency semantics. In *ESOP*. Springer, Berlin, Heidelberg, 283–307. https://doi.org/10.1007/978-3-662-46669-8_12

[14] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ concurrency. In *POPL*. ACM, New York, NY, USA, 55–66. https://doi.org/10.1145/1925844.1926394

[15] Giovanni Bernardi and Alexey Gotsman. 2016. Robustness against consistency models with atomic visibility. In *CONCUR*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 7:1–7:15. https://doi.org/10.4230/LIPIcs.CONCUR.2016.7

[16] Ahmed Bouajjani, Constantin Enea, Rachid Guerraoui, and Jad Hamza. 2017. On verifying causal consistency. In *POPL*. ACM, New York, NY, USA, 626–638. https://doi.org/10.1145/3009837.3009888

[17] Lucas Brutschy, Dimitar Dimitrov, Peter Müller, and Martin Vechev. 2018. Static serializability analysis for causal consistency. In *PLDI*. ACM, New York, NY, USA, 90–104. https://doi.org/10.1145/3192366.3192415

[18] Sebastian Burckhardt. 2014. Principles of eventual consistency. *Found. Trends Program. Lang.* 1, 1-2 (Oct. 2014), 1–150. https://doi.org/10.1561/2500000011

[19] Andrea Cerone, Alexey Gotsman, and Hongseok Yang. 2015. Transaction chopping for parallel snapshot isolation. In *DISC*. Springer-Verlag, Berlin, Heidelberg, 388–404. https://doi.org/10.1007/978-3-662-48653-5_26

[20] Andrea Cerone, Alexey Gotsman, and Hongseok Yang. 2017. Algebraic laws for weak consistency. In *CONCUR*, Vol. 85. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 26:1–26:18. https://doi.org/10.4230/LIPIcs.CONCUR.2017.26

[21] Simon Doherty, Brijesh Dongol, Heike Wehrheim, and John Derrick. 2019. Verifying C11 programs operationally. In *PPoPP*. ACM, New York, NY, USA, 355–365. https://doi.org/10.1145/3293883.3295702

[22] Alain Finkel and Philippe Schnoebelen. 2001. Well-structured transition systems everywhere! *Theoretical Computer Science* 256, 1 (2001), 63 – 92. https://doi.org/10.1016/S0304-3975(00)00102-X

[23] ISO/IEC 14882:2011. 2011. Programming language C++.

[24] ISO/IEC 9899:2011. 2011. Programming language C.

[25] Jan-Oliver Kaiser, Hoang-Hai Dang, Derek Dreyer, Ori Lahav, and Viktor Vafeiadis. 2017. Strong logic for weak memory: Reasoning about release-acquire consistency in Iris. In *ECOOP*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 17:1–17:29. https://doi.org/10.4230/LIPIcs.ECOOP.2017.17

[26] Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. 2017. A Promising Semantics for Relaxed-Memory Concurrency. In *POPL*. ACM, New York, NY, USA, 175–189. https://doi.org/10.1145/3009837.3009850

[27] Michalis Kokologiannakis, Ori Lahav, Konstantinos Sagonas, and Viktor Vafeiadis. 2017. Effective stateless model checking for C/C++ concurrency. *Proc. ACM Program. Lang.* 2, POPL, Article 17 (Dec. 2017), 32 pages. https://doi.org/10.1145/3158105

[28] Michael Kuperstein, Martin Vechev, and Eran Yahav. 2011. Partial-coherence abstractions for relaxed memory models. In *PLDI*. ACM, New York, NY, USA, 187–198. https://doi.org/10.1145/1993498.1993521

[29] Ori Lahav. 2019. Verification under causally consistent shared memory. *ACM SIGLOG News* 6, 2 (April 2019), 43–56. https://doi.org/10.1145/3326938.3326942

[30] Ori Lahav, Nick Giannarakis, and Viktor Vafeiadis. 2016. Taming release-acquire consistency. In *POPL*. ACM, New York, NY, USA, 649–662. https://doi.org/10.1145/2837614.2837643

[31] Ori Lahav and Roy Margalit. 2019. Robustness against release/acquire semantics. In *PLDI*. ACM, New York, NY, USA, 126–141. https://doi.org/10.1145/3314221.3314604

[32] Ori Lahav and Viktor Vafeiadis. 2015. Owicki-Gries reasoning for weak memory models. In *ICALP*. Springer-Verlag, Berlin, Heidelberg, 311–323. https://doi.org/10.1007/978-3-662-47666-6_25

[33] Ori Lahav and Viktor Vafeiadis. 2016. Explaining relaxed memory models with program transformations. In *FM*. Springer, Cham, 479–495. https://doi.org/10.1007/978-3-319-48989-6_29

[34] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. 2017. Repairing sequential consistency in C/C++11. In *PLDI*. ACM, New York, NY, USA, 618–632. https://doi.org/10.1145/3062341.3062352

[35] Mohsen Lesani, Christian J. Bell, and Adam Chlipala. 2016. Chapar: Certified causally consistent distributed key-value stores. In *POPL*. ACM, New York, NY, USA, 357–370. https://doi.org/10.1145/2837614.2837622

[36] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. 2012. Making geo-replicated systems fast as possible, consistent when necessary. In *OSDI*. USENIX Association, Berkeley, CA, USA, 265–278. http://dl.acm.org/citation.cfm?id=2387880.2387906

[37] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. 2011. Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS. In *SOSP*. ACM, New York, NY, USA, 401–416. https://doi.org/10.1145/2043556.2043593

[38] Mapping 2019. C/C++11 mappings to processors. Retrieved July 3, 2019 from http://www.cl.cam.ac.uk/~pes20/cpp/cpp0xmappings.html

[39] Luc Maranget, Susmit Sarkar, and Peter Sewell. 2012. A tutorial introduction to the ARM and POWER relaxed memory models. http://www.cl.cam.ac.uk/~pes20/ppc-supplemental/test7.pdf.

[40] MongoDB Manual 4.2 2019. Causal consistency and read and write concerns. Retrieved November 19, 2019 from https://docs.mongodb.com/manual/core/causal-consistency-read-write-concerns

[41] Kartik Nagar and Suresh Jagannathan. 2018. Automated detection of serializability violations under weak consistency. In *CONCUR*, Vol. 118. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 41:1–41:18. https://doi.org/10.4230/LIPIcs.CONCUR.2018.41

[42] Scott Owens, Susmit Sarkar, and Peter Sewell. 2009. A better x86 memory model: x86-TSO. In *TPHOLs*. Springer, Heidelberg, 391–407. https://doi.org/10.1007/978-3-642-03359-9_27

[43] Azalea Raad, Ori Lahav, and Viktor Vafeiadis. 2018. On parallel snapshot isolation and release/acquire consistency. In *ESOP*. Springer, Berlin, Heidelberg, 940–967. https://doi.org/10.1007/978-3-319-89884-1_33

[44] Sylvain Schmitz and Philippe Schnoebelen. 2012. Algorithmic aspects of WQO theory. (Aug. 2012). https://cel.archives-ouvertes.fr/cel-00727025 Lecture notes.

[45] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. 2011. Transactional storage for geo-replicated systems. In *SOSP*. ACM, New York, NY, USA, 385–400. https://doi.org/10.1145/2043556.2043592

[46] Thibault Suzanne and Antoine Miné. 2016. From array domains to abstract interpretation under store-buffer-based memory models. In *SAS*. Springer, Berlin, Heidelberg, 469–488.

[47] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike Spreitzer, Marvin Theimer, and Brent W. Welch. 1994. Session guarantees for weakly consistent replicated data. In *PDIS*. IEEE Computer Society, Washington, DC, USA, 140–149. http://dl.acm.org/citation.cfm?id=645792.668302

[48] Aaron Turon, Viktor Vafeiadis, and Derek Dreyer. 2014. GPS: Navigating weak memory with ghosts, protocols, and separation. In *OOPSLA*. ACM, New York, NY, USA, 691–707. https://doi.org/10.1145/2660193.2660243

[49] Viktor Vafeiadis and Chinmay Narayan. 2013. Relaxed separation logic: A program logic for C11 concurrency. In *OOPSLA*. ACM, New York, NY, USA, 867–884. https://doi.org/10.1145/2509136.2509532

[50] John Wickerson, Mark Batty, Tyler Sorensen, and George A. Constantinides. 2017. Automatically comparing memory consistency models. In *POPL*. ACM, New York, NY, USA, 190–204. https://doi.org/10.1145/3009837.3009838