Decidable Verification under Localized Release-Acquire Concurrency

Abhishek Kr Singh(✉) and Ori Lahav

Tel Aviv University, Tel Aviv, Israel
abhishek.uor@gmail.com, orilahav@tau.ac.il

Abstract. State reachability for finite state concurrent programs running under Release-Acquire (RA) semantics is known to be undecidable, while under a weaker variant, called Weak-Release-Acquire (WRA), the problem is decidable. However, WRA allows many counterintuitive behaviors not allowed under RA, in which threads locally oscillate between observed values. We propose a strengthening of WRA in the form of a new memory model, which we call Localized Release-Acquire (LRA), that prunes these oscillatory behaviors. We provide semantics for LRA and show that verification under LRA is decidable by extending the potential-based technique used to prove decidability under WRA. The LRA model is still weaker than RA, and thus our results can be used to soundly verify programs under RA.

Keywords: Relaxed Memory Concurrency · State Reachability · Release-Acquire Semantics

1 Introduction

The *Release-Acquire* memory model (RA), a prominent fragment of the C/C++ shared-memory concurrency specifications from 2011 [13, 16, 17, 27], has recently gained a lot of attention (see, e.g., [2, 7, 18, 23–25, 30]). For programmers, RA combines the essential guarantees of coherence [11] (a.k.a. "sequential consistency per-location") and causal consistency [10, 20], which enable the implementation of various concurrent algorithms and synchronization mechanisms with very few barriers. For implementors, RA is weaker than the Total Store Order model (TSO) [29, 31], which enables efficient mapping of memory accesses to Intel's x86 processors. Moreover, unlike TSO, RA is "monotone" [33], which, roughly speaking, means that replacing parallel composition with sequential composition can never introduce additional behaviors [26].

Unfortunately, the fundamental problem of state reachability in finite-state concurrent programs running under RA was recently shown to be undecidable [2]. This is in contrast with state reachability assuming the well-known model of sequential consistency (SC) [28], which amounts to standard reachability in a finite state system, as well as with state reachability assuming TSO, which was shown to be decidable [4, 5, 12] using the framework of well-structured transition systems (WSTS) [1, 15]. More recently, decidability of state reachability was

B. Finkbeiner and L. Kovács (Eds.): TACAS 2024, LNCS 14572, pp. 235–254, 2024.
https://doi.org/10.1007/978-3-031-57256-2_12

established for two variants of RA [21,22], called Strong Release-Acquire (SRA) and Weak Release-Acquire (WRA), which bound RA from above (every behavior allowed by SRA is allowed by RA) and below (every behavior allowed by RA is allowed by WRA). In particular, verification under WRA can be used to obtain sound (but incomplete) verification under RA, since any buggy program under RA is also buggy under WRA. The gap, however, between WRA and RA includes some dubious behaviors:

Example 1. The annotated behaviors in the three litmus tests below are allowed by WRA but disallowed by RA:

```
      (Oscillation 1)     |          (Oscillation 2)          |            (Oscillation 3)
        ‖ x := 1          |        ‖ a := x  //1 ‖            |        ‖ a := y  //1 ‖
x := 2  ‖ b := x  //2     | x := 2 ‖ b := x  //2 ‖ x := 1     | x := 2 ‖ b := x  //2 ‖ x := 1
        ‖ c := x  //1     |        ‖ c := x  //1 ‖            |        ‖ c := x  //1 ‖ y := 1
```

Intuitively speaking, a thread in WRA can "change its mind" about the order of concurrent writes. In RA, every shared variable is governed by a "modification order" which dictates the (globally agreed upon) order of concurrent writes, and reads have to respect that order.

In this paper, we aim to narrow the gap between models with decidable reachability problem and RA by providing a model that lies between WRA and RA and still allows for decidable verification. More concretely, we propose to strengthen WRA in a way that eliminates the above oscillatory behaviors, while still (1) being weaker than RA and (2) inducing a decidable state reachability problem. The proposed model, which we call Localized Release-Acquire (LRA), is obtained by adding one constraint (a.k.a. axiom) to WRA's declarative consistency predicate. In turn, decidability is established similarly to [22], by carefully designing an operational "lossy" semantics based on maintaining *thread potentials*, so that it fits well in the framework of WSTS, and it is equivalent to LRA. Our proof establishes the equivalence of the lossy potential-based system with LRA using forward simulation in one direction and backward simulation in the converse.

The full version of this paper available in [32] contains detailed proofs for the claims of the paper.

2 Preliminaries

In this section we present the formal preliminaries for our results, including the representation of concurrent programs, memory systems, and declarative execution graphs. We employ the following *finite* domains (and metavariables ranging over them):

$$\begin{aligned} \textit{thread identifiers } \tau, \pi &\in \mathsf{Tid} = \{\mathtt{T}_1, \mathtt{T}_2, ...\} \\ \textit{variables } x, y &\in \mathsf{Loc} \triangleq \{\mathtt{x}, \mathtt{y}, ...\} \\ \textit{values } v &\in \mathsf{Val} \triangleq \{0, 1, 2, ...\} \end{aligned}$$

We represent concurrent programs as labeled transition systems. A *labeled transition system* (LTS, for short) 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 $A.\mathtt{Q}$, $A.\mathtt{Q}_0$, and $A.\mathtt{T}$ the three components of an LTS A; we write $\xrightarrow{\sigma}_A$ for the relation $\{\langle q, q' \rangle \mid \langle q, \sigma, q' \rangle \in A.\mathtt{T}\}$ and $\rightarrow_A$ for $\bigcup_{\sigma \in \Sigma} \xrightarrow{\sigma}_A$. A state $q \in A.\mathtt{Q}$ is *reachable* in A if $q_0 \rightarrow_A^* q$ for some $q_0 \in A.\mathtt{Q}_0$. A sequence $\sigma_1, \ldots, \sigma_n$ is a *trace* of A if $q_0 \xrightarrow{\sigma_1}_A q_1 \xrightarrow{\sigma_2}_A \cdots q_{n-1} \xrightarrow{\sigma_n}_A q_n$ for some $q_0 \in A.\mathtt{Q}_0$ and $q_1, \ldots, q_n \in A.\mathtt{Q}$.

For brevity, we elide the definition of how concurrent programs in a programming language are interpreted as LTSs (see [22] for such definition), but only note that these LTSs are *finite-state* and they employ labels (a.k.a. "program transition labels") from the set $\mathsf{ProgLab} \triangleq \mathsf{Tid} \times (\mathsf{Lab} \cup \{\epsilon\})$, where Lab denotes the set of *action labels*, representing interactions that a program may have with the memory, and ϵ denotes a thread-internal transition. Action labels $l \in \mathsf{Lab}$ take one of the following forms: a read $\mathtt{R}(x, v_{\mathtt{R}})$, a write $\mathtt{W}(x, v_{\mathtt{W}})$, or a read-modify-write $\mathtt{RMW}(x, v_{\mathtt{R}}, v_{\mathtt{W}})$, where $x \in \mathsf{Loc}$ and $v_{\mathtt{R}}, v_{\mathtt{W}}, \in \mathsf{Val}$. The functions $\mathtt{typ}$, $\mathtt{loc}$, $\mathtt{val}_{\mathtt{R}}$, and $\mathtt{val}_{\mathtt{W}}$ respectively retrieve (when applicable) the type ($\mathtt{R}/\mathtt{W}/\mathtt{RMW}$), variable ($x$), read value ($v_{\mathtt{R}}$), and written value ($v_{\mathtt{W}}$) of an action label. Furthermore, for a program transition label $\alpha \in \mathsf{ProgLab}$, the functions $\mathtt{tid}$ and $\mathtt{lab}$ respectively retrieve the thread identifier (τ) and the action label (or ϵ) of α, and the functions on action labels ($\mathtt{typ}$, $\mathtt{loc}$, ...) are lifted to program transition labels in the obvious way.

To represent concurrent programs running under a particular memory model, we synchronize the transitions of a program Pr with a memory system. A memory system is another LTS $\mathcal{M}$ (but, possibly infinite-state) whose set of transition labels consists of non-silent program transition labels (elements of $\mathsf{Tid} \times \mathsf{Lab}$) as well as a (disjoint) set $\mathcal{M}.\boldsymbol{\Theta}$ of memory-internal actions. Then, the composition of a program Pr and a memory system $\mathcal{M}$, denoted by $Pr \bowtie \mathcal{M}$, is the LTS whose transition labels are the elements of $\mathsf{ProgLab} \cup \mathcal{M}.\boldsymbol{\Theta}$; states are pairs $\langle \overline{p}, M \rangle \in Pr.\mathtt{Q} \times \mathcal{M}.\mathtt{Q}$; initial state is $\langle \overline{p}_{\mathsf{Init}}, \mathcal{M}.\mathtt{Q}_0 \rangle$; and transitions are given by:

$$\frac{\alpha \in \mathsf{Tid} \times \mathsf{Lab} \qquad \overline{p} \xrightarrow{\alpha}_{Pr} \overline{p}' \qquad M \xrightarrow{\alpha}_{\mathcal{M}} M'}{\langle \overline{p}, M \rangle \xrightarrow{\alpha}_{Pr \bowtie \mathcal{M}} \langle \overline{p}', M' \rangle} \qquad \frac{\alpha \in \mathsf{Tid} \times \{\epsilon\} \qquad \overline{p} \xrightarrow{\alpha}_{Pr} \overline{p}'}{\langle \overline{p}, M \rangle \xrightarrow{\alpha}_{Pr \bowtie \mathcal{M}} \langle \overline{p}', M \rangle} \qquad \frac{\alpha \in \mathcal{M}.\boldsymbol{\Theta} \qquad M \xrightarrow{\alpha}_{\mathcal{M}} M'}{\langle \overline{p}, M \rangle \xrightarrow{\alpha}_{Pr \bowtie \mathcal{M}} \langle \overline{p}, M' \rangle}$$

The state reachability problem for a memory system $\mathcal{M}$ receives as input a program Pr and a state $\overline{p} \in Pr.\mathtt{Q}$ and asks whether $\langle \overline{p}, M \rangle$ is reachable in $Pr \bowtie \mathcal{M}$ for some $M \in \mathcal{M}.\mathtt{Q}$.

Finally, we also need the notion of a *declarative* memory model, which accepts/rejects program behaviors based on constraints on the generated *execution graphs*.

Definition 1. An *execution graph* G is a pair $\langle E, rf \rangle$, where:

- E is a finite set of *events*. An *event* e is a tuple $\langle \tau, s, l \rangle$, where $\tau \in \mathsf{Tid}$, called the event's *thread identifier*; $s \in \mathbb{N}$, called the event's *serial identifier*, and $l \in \mathsf{Lab}$, called the event's *label*. The functions $\mathtt{tid}$, $\mathtt{sn}$, and $\mathtt{lab}$ return the thread identifier (τ), identifier (s), and action label (l) of an event. All functions on action labels (typ, loc, ...) are lifted to events in the obvious way. We denote by E the set of all events, and define the following subsets:

$$\mathsf{R} \triangleq \{e \in \mathsf{E} \mid \mathtt{typ}(e) \in \{\mathtt{R}, \mathtt{RMW}\}\} \qquad \mathsf{W} \triangleq \{e \in \mathsf{E} \mid \mathtt{typ}(e) \in \{\mathtt{W}, \mathtt{RMW}\}\}$$
$$\mathsf{RMW} \triangleq \mathsf{R} \cap \mathsf{W} \qquad \mathsf{E}^{\tau} = \{e \in \mathsf{E} \mid \mathtt{tid}(e) = \tau\}$$

– rf is a *reads-from relation for* E, that is a relation on E satisfying:
  - If $\langle w, r\rangle \in rf$, then $w \in \mathsf{W}$ and $r \in \mathsf{R}$.
  - If $\langle w, r\rangle \in rf$, then $\mathtt{loc}(w) = \mathtt{loc}(r)$ and $\mathtt{val}_{\mathtt{W}}(w) = \mathtt{val}_{\mathtt{R}}(r)$.
  - $w_1 = w_2$ whenever $\langle w_1, r\rangle, \langle w_2, r\rangle \in rf$ (each read reads from at most one write).
  - For every $r \in E \cap \mathsf{R}$, there exists some $w \in E$ such that $\langle w, r\rangle \in rf$ (each read reads from some write).

We denote the components of G by $G.\mathtt{E}$ and $G.\mathtt{rf}$. For any set $E' \subseteq \mathsf{E}$, we write $G.E'$ for $G.\mathtt{E} \cap E'$ (e.g., $G.\mathsf{W} = G.\mathtt{E} \cap \mathsf{W}$). The *program order* induced by an execution graph G, denoted by $G.\mathsf{po}$, is defined as $G.\mathsf{po} \triangleq \{\langle e_1, e_2\rangle \in E \times E \mid \mathtt{sn}(e_1) < \mathtt{sn}(e_2) \wedge \mathtt{tid}(e_1) = \mathtt{tid}(e_2)\}$.

Given a set E of events, $\tau \in \mathsf{Tid}$, and $l \in \mathsf{Lab}$, $\mathsf{NextEvent}(E, \tau, l)$ denotes the event with thread identifier τ, label l, and a minimal fresh serial identifier w.r.t. E, i.e., $\mathsf{NextEvent}(E, \tau, l) \triangleq \langle \tau, s, l\rangle$, where $s = \min\{n \in \mathbb{N} \mid \langle \tau, n, l\rangle \notin E\}$.

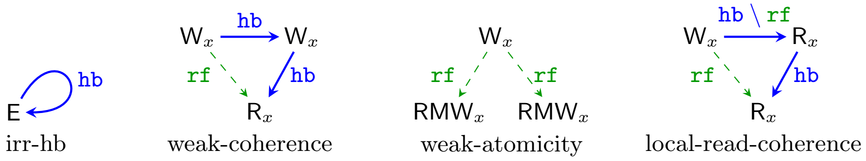
Definition 2. An execution graph G is *generated by* a program Pr with final state $\overline{p} \in Pr.\mathtt{Q}$ if $\langle \overline{p}_0, G_0\rangle \to^* \langle \overline{p}, G\rangle$ for some $\overline{p}_0 \in Pr.\mathtt{Q}_0$, where G_0 denotes the empty execution graph (given by $G_0 \triangleq \langle \emptyset, \emptyset\rangle$) and $\to$ is defined by:

$$\frac{\overline{p} \xrightarrow{\tau, l}_{Pr} \overline{p}' \qquad E' = E \cup \{\mathsf{NextEvent}(E, \tau, l)\} \qquad rf \subseteq rf' \qquad \langle E', rf'\rangle \text{ is an execution graph}}{\langle \overline{p}, \langle E, rf\rangle\rangle \to \langle \overline{p}', \langle E', rf'\rangle\rangle} \qquad \frac{\overline{p} \xrightarrow{\tau, \varepsilon}_{Pr} \overline{p}'}{\langle \overline{p}, G\rangle \to \langle \overline{p}', G\rangle}$$

Using the above definitions, a declarative memory model can be identified with a set of so-called *consistent* execution graphs, and a program state $\overline{p}$ is 'emphreachable under a declarative memory model if some consistent execution graph G is generated by Pr with final state $\overline{p}$.

3 The Localized Release-Acquire Model

In this section we introduce the Localized Release-Acquire (LRA) model, starting with its declarative presentation. LRA is obtained by adding a single constraint, called "local-read-coherence", to WRA. We first briefly repeat the three constraints of WRA (see [20] for more details). Figure 1 summarizes the four constraints of LRA.



irr-hb weak-coherence weak-atomicity local-read-coherence

Fig. 1. Illustration of forbidden patterns in LRA

Notation for 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 $R_1 \mathbin{;} R_2$. We denote by $[A]$ the identity relation on a set A (e.g., $[A] \mathbin{;} R \mathbin{;} [B] = R \cap (A \times B)$).

First, we need a derived "happens-before" relation. For a given execution graph G, we define $G.\texttt{hb} \triangleq (G.\mathsf{po} \cup G.\texttt{rf})^+$. We require that $G.\texttt{hb}$ is a partial order, which results in our first constraint:

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

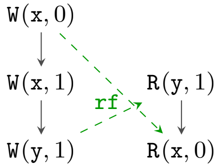
The next constraint intuitively makes sure that "a thread cannot read a value when it is aware of a later value written to the same location", where "aware" and "later" are interpreted using $G.\texttt{hb}$. Formally, we define $G.\texttt{hb}|_{\texttt{loc}} \triangleq \{\langle e_1, e_2\rangle \in G.\texttt{hb} \mid \texttt{loc}(e_1) = \texttt{loc}(e_2)\}$ (i.e., per-location restriction of the happens-before relation), and require the following:

$$G.\texttt{hb}|_{\texttt{loc}} \mathbin{;} [\mathsf{W}] \mathbin{;} G.\texttt{hb} \mathbin{;} G.\texttt{rf}^{-1} \text{ is irreflexive} \qquad \text{(weak-coherence)}$$

In particular, the following annotated outcome of the message-passing (MP) test is forbidden:

(MP)

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



An execution graph justifying this outcome must have $\texttt{rf}$-edges as depicted above. However, we have $\texttt{hb}|_{\texttt{loc}}$ from $\texttt{W}(\texttt{x}, 0)$ to $\texttt{W}(\texttt{x}, 1)$, $\texttt{hb}$ from $\texttt{W}(\texttt{x}, 1)$ to $\texttt{R}(\texttt{x}, 0)$, and $\texttt{rf}$ from $\texttt{W}(\texttt{x}, 0)$ to $\texttt{R}(\texttt{x}, 0)$, which is forbidden by weak-coherence.

The final condition that comes from WRA ensures that distinct RMW events never read from the same write event:

$$\forall \langle w_1, e_1\rangle, \langle w_2, e_2\rangle \in G.\texttt{rf} \mathbin{;} [\mathsf{RMW}].\ w_1 = w_2 \implies e_1 = e_2 \qquad \text{(weak-atomicity)}$$

This concludes the consistency constraints of WRA. As noted above, unlike RA, WRA admits behaviors in which threads oscillate between values that were concurrently written to the same location. Our proposed condition of LRA that prunes these behaviors is the following:

$$(G.\texttt{hb}|_{\texttt{loc}} \setminus G.\texttt{rf}) \mathbin{;} [\mathsf{R}] \mathbin{;} G.\texttt{hb} \mathbin{;} G.\texttt{rf}^{-1} \text{ is irreflexive} \qquad \text{(local-read-coherence)}$$

Intuitively, this constraint ensures that a thread cannot read from a certain write w if it is already aware of a read r' reading from the same location that is later than w and reads from some other write w'. Again, “aware” and “later” are interpreted using $G.\mathtt{hb}$.

The following examples demonstrate “oscillations” between observed values that are allowed in WRA but forbidden in LRA.

(Oscillation 1)

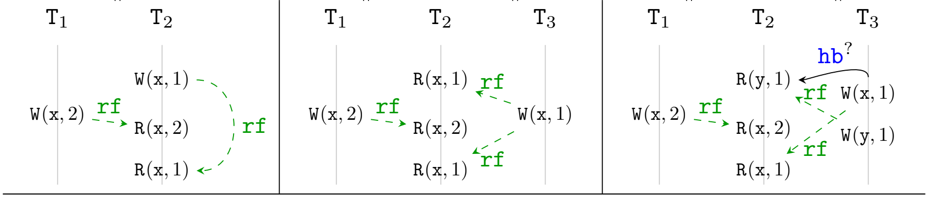
```
x := 2 || x := 1
          b := x  //2
          c := x  //1
```

(Oscillation 2)

```
x := 2 || a := x  //1 || x := 1
          b := x  //2
          c := x  //1
```

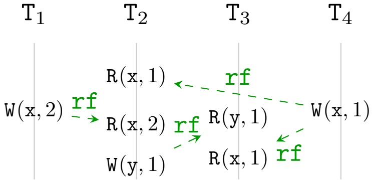
(Oscillation 3)

```
x := 2 || a := y  //1 || x := 1
          b := x  //2    y := 1
          c := x  //1
```



(Order-Propagation)

```
x := 2 || a := x  //1 || d := y  //1 || x := 1
          b := x  //2    c := x  //1
          y := 1
```



It can be checked that local-read-coherence forbids these execution graphs: in all of them we have (1) $G.\mathtt{hb}|_{\mathtt{loc}} \setminus G.\mathtt{rf}$ from $\mathtt{W}(\mathtt{x}, 1)$ to $\mathtt{R}(\mathtt{x}, 2)$; (2) $G.\mathtt{hb}$ from $\mathtt{R}(\mathtt{x}, 2)$ to the read $\mathtt{R}(\mathtt{x}, 1)$ that represents the read to $\mathtt{c}$; and (3) $\mathtt{rf}$ from $\mathtt{W}(\mathtt{x}, 1)$ to that read.

Next, we establish the relation between LRA and RA (see [22] for a definition of RA).

Proposition 1. LRA is weaker than RA, that is: if a program state is reachable under RA, then it is also reachable under LRA.

Proof. We establish this result by recalling the following “read-coherence” consistency constraint of RA (see Figure 2 and [20] for more details). Note the use of modification order $G.\mathtt{mo}$ in RA to interpret one write being “later” than another, in the place of $G.\mathtt{hb}|_{\mathtt{loc}}$ in the “weak-coherence” in WRA. Here $G.\mathtt{mo}$ is disjoint union of relations $\{G.\mathtt{mo}_x\}_{x \in \mathsf{Loc}}$ where each $G.\mathtt{mo}_x$ is a strict total order on W_x.

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

Since WRA is strictly weaker than RA, it suffices to show that the additional constraint “local-read-coherence” of LRA is also guaranteed in RA. The proof follows by contradiction. Assume otherwise, hence, for a given $x \in \mathsf{Loc}$, we have $w, w' \in \mathsf{W}_x$ and $r, r' \in \mathsf{R}_x$ where $\langle w, r' \rangle \in \mathtt{hb} \setminus \mathtt{rf}$, $\langle w', r' \rangle \in \mathtt{rf}$, $\langle w, r \rangle \in \mathtt{rf}$, and $w \neq w'$ (see right side of Figure 2). Since $\mathtt{loc}(w) = x = \mathtt{loc}(w')$, due to the RA semantics, we have one of the following cases:

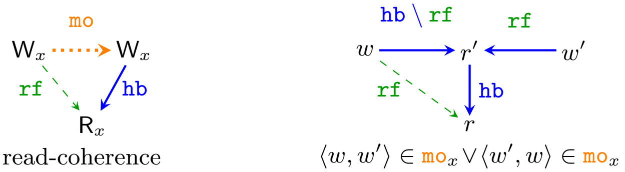


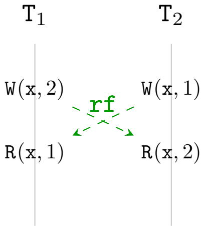
Fig. 2. Axiom read-coherence in RA and illustration for proof of Proposition 1

- $\langle w, w' \rangle \in \texttt{mo}_x$: In this case we have $\langle w, r \rangle \in \texttt{rf}$ while $\langle w, r \rangle \in \texttt{mo}_x \,;\texttt{hb}$, which contradicts the axiom read-coherence of RA.
- $\langle w', w \rangle \in \texttt{mo}_x$: In this case we have $\langle w', r' \rangle \in \texttt{rf}$ while $\langle w', r' \rangle \in \texttt{mo}_x \,;\texttt{hb}$, which again contradicts the axiom read-coherence of RA.

To see that LRA is *strictly* weaker than RA, we note that LRA does not provide full coherence. Indeed, as the next example shows, even programs with a single shared variable can exhibit weak behaviors:

(WW)

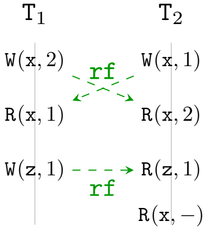
```
x := 2          ||  x := 1
a := x   // 1   ||  b := x   // 2
```



Interestingly, our final example shows the LRA model is possibly blocking: it may be the case that a thread simply cannot read from a certain location, since any option for reading would violate local-read-coherence.

(Blocking)

```
x := 2          ||  x := 1
a := x   // 1   ||  b := x   // 2
z := 1          ||  c := z   // 1
                ||  d := x   // nothing can be read
```



Roughly speaking, the synchronization on z "joins" the threads and rules out both options. More formally, if the final read reads from W(x, 1), we violate local-read-coherence due to $G.\texttt{hb}|_{\texttt{loc}} \setminus G.\texttt{rf}$ from W(x, 1) to R(x, 2) and $G.\texttt{hb}$ from R(x, 2) to the final read. In turn, if the final read reads from W(x, 2), we violate local-read-coherence due to $G.\texttt{hb}|_{\texttt{loc}} \setminus G.\texttt{rf}$ from W(x, 2) to R(x, 1) and $G.\texttt{hb}$ from R(x, 1) to the final read.

It is important to note that the blocking aspect of LRA model does not affect the benefits of sound verification of the RA programs using LRA, since (due to Proposition 1) forbidden outcomes in LRA model (possibly due to a blocked run) are also forbidden in the RA model.

3.1 An Operational Presentation

Since LRA-consistency is “prefix-closed”, it is straightforward to “operationalize” LRA’s declarative presentation, which will help us below in relating the potential-model to LRA. To do so, we define a memory system, called opLRA, whose states are execution graphs, the only initial state is the empty execution graph, and the transitions are as follows:

$$\textsc{write}\quad \dfrac{\begin{array}{c} e = \mathsf{NextEvent}(G.\mathtt{E}, \tau, \mathtt{W}(x, v_{\mathtt{W}})) \\ G' = \langle G.\mathtt{E} \cup \{e\}, G.\mathtt{rf} \rangle \end{array}}{G \xrightarrow{\tau, \mathtt{W}(x, v_{\mathtt{W}})}_{\text{opLRA}} G'}$$

$$\textsc{read/rmw}\quad \dfrac{\begin{array}{c} l = \mathtt{R}(x, v_{\mathtt{R}}) \lor l = \mathtt{RMW}(x, v_{\mathtt{R}}, v_{\mathtt{W}}) \\ e = \mathsf{NextEvent}(G.\mathtt{E}, \tau, l) \qquad G' = \langle G.\mathtt{E} \cup \{e\}, G.\mathtt{rf} \cup \{\langle w, e \rangle\} \rangle \\ w \in G.\mathsf{W}_x \qquad \mathtt{val}_{\mathtt{W}}(w) = v_{\mathtt{R}} \\ w \notin \mathit{dom}(G.\mathtt{hb}|_{\mathtt{loc}} ; [\mathsf{W}] ; G.\mathtt{hb}^? ; [\mathsf{E}^\tau]) \\ w \notin \mathit{dom}((G.\mathtt{hb}|_{\mathtt{loc}} \setminus G.\mathtt{rf}) ; [\mathsf{R}] ; G.\mathtt{hb}^? ; [\mathsf{E}^\tau]) \\ \mathtt{typ}(l) = \mathtt{RMW} \implies w \notin \mathit{dom}(G.\mathtt{rf} ; [\mathsf{RMW}]) \end{array}}{G \xrightarrow{\tau, l}_{\text{opLRA}} G'}$$

These transitions are enforcing consistency on every step, which allows us to establish the following relation.

Proposition 2. LRA is equivalent to opLRA, that is: a program state is reachable under LRA iff it is reachable under opLRA.

4 Lossy semantics for LRA

In this section, we present loLRA, a potential-based memory system that is equivalent to LRA and well suited for verification in the framework of WSTS.

The memory states of loLRA maintain a collection of "read/write-option" lists for each thread, called the *potential* of the thread. Concretely, a state of loLRA is a *potential mapping* $\mathcal{B}$ which maps each thread $\tau \in \mathsf{Tid}$ to its potential $\mathcal{B}(\tau)$. Potentials are finite sets of *option lists*, where each option list stands for a sequence of possible future reads (*read options*) and writes (*write options*) that ascribe possible operations the thread may perform in the order it may perform them. For instance, a list $o_1 \cdot o_2$ consisting of two read options, o_1 and 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 $\mathtt{loc}(o_2)$. Thread potentials are explicitly “lossy”—a thread can non-deterministically lose whatever parts of its potential at any point. Initially, the loLRA memory system non-deterministically starts in a state where all potentials consist solely of write options.

Next, we present the full definitions (which, except for loLRA’s transitions match precisely the definitions of the corresponding system for WRA in [22]).

Notation for 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 a sequence s over Σ with its underlying function in $\{1, ..., |s|\} \to \Sigma$, and write $s(k)$ for the symbol at position $1 \leq k \leq |s|$ in s. We write $\sigma \in s$ if the symbol 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, and lift it to concatenation of sets S_1 and S_2 of sequences in the obvious way ($S_1 \cdot S_2 \triangleq \{s_1 \cdot s_2 \mid s_1 \in S_1, s_2 \in S_2\}$). We identify symbols with sequences of length 1 or their singletons when needed (e.g., in expressions like $\sigma \cdot S$ for $\sigma \in \Sigma$ and a set S of sequences over Σ).

Definition 3. Options, option lists, potentials, and potential mappings are defined as follows:

1. An *option* o is either $\langle \tau, x, v, \pi_{\texttt{RMW}} \rangle$ (*read option*) or $\texttt{O}_{\texttt{W}}(x)$ (*write option*), where $\tau, \pi_{\texttt{RMW}} \in \mathsf{Tid}$, $x \in \mathsf{Loc}$, and $v \in \mathsf{Val}$. The functions $\texttt{typ}$, $\texttt{tid}$, $\texttt{loc}$, $\texttt{val}$, and $\texttt{rmw-tid}$ return (when applicable) the type ($\texttt{R}/\texttt{W}$), thread identifier (τ), location (x), value (v), and RMW thread identifier ($\pi_{\texttt{RMW}}$) of a given option.
2. An *option list* L is a finite sequence of (read or write) options. For a given option list L, we define $\texttt{loc}(L) \triangleq \{\texttt{loc}(o) \mid o \in L\}$.
3. A *potential* B is a finite non-empty set of option lists.
4. A *potential mapping* $\mathcal{B}$ is a function assigning a potential to every $\tau \in \mathsf{Tid}$.

We define a (well quasi) ordering on option lists that naturally extends to potentials and to potential mappings.

Definition 4. The (overloaded) relation $\sqsubseteq$ is defined by:

1. on 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 potential mappings: $\mathcal{B} \sqsubseteq \mathcal{B}'$ if $\mathcal{B}(\tau) \sqsubseteq \mathcal{B}'(\tau)$ for every $\tau \in \mathsf{Tid}$ (componentwise order).

The memory system loLRA is formally defined as follows.

Definition 5. The memory system loLRA is defined by:

- loLRA.$\texttt{Q}$ is the set of potential mappings.
- loLRA.$\texttt{Q}_0 = \{\mathcal{B} \mid \forall \tau \in \mathsf{Tid}, L \in \mathcal{B}(\tau), o \in L.\ \texttt{typ}(o) = \texttt{W}\}$.
- The transitions of loLRA are given in Figure 3.

The transitions of loLRA are informally understood as follows:

- READ: For a thread τ to read v from x, all lists of τ should start with an option o with $\texttt{val}(o) = v$ and $\texttt{loc}(o) = x$ (since it is the same option o in the head of all lists, all lists of τ also start with the same thread identifier, which is important for the equivalence result; see [22, Example 5.5]). The read step consumes these options by discarding the first element from each of τ's lists.

WRITE

$$\frac{\begin{array}{l} \qquad\qquad\qquad\qquad\qquad o = \langle \tau, x, v_{\mathtt{W}}, \pi_{\mathtt{RMW}} \rangle \\ \forall \pi \in \mathsf{Tid}, L' \in \mathcal{B}'(\pi). \\ \quad ((\pi = \tau \implies \mathtt{O}_{\mathtt{W}}(x) \cdot L' \in \mathcal{B}(\tau)) \land (\pi \neq \tau \implies L' \in \mathcal{B}(\pi))) \;\lor \\ \quad (\exists n \geq 1, L_0, \dots, L_n. \\ \qquad L' = L_0 \cdot (o \cdot L_1) \cdot (o \cdot L_2) \cdot \dots \cdot (o \cdot L_n) \;\land \\ \qquad \mathtt{O}_{\mathtt{W}}(x) \cdot (L_1 \cdot \dots \cdot L_{n-1}) \cdot \mathtt{O}_{\mathtt{W}}(x) \cdot L_n \in \mathcal{B}(\tau) \;\land \\ \qquad (\pi = \tau \implies \mathtt{O}_{\mathtt{W}}(x) \cdot L_0 \cdot \dots \cdot L_{n-1} \cdot \mathtt{O}_{\mathtt{W}}(x) \cdot L_n \in \mathcal{B}(\tau) \land x \notin \mathtt{loc}(L_0 \cdot \dots \cdot L_{n-1})) \;\land \\ \qquad (\pi \neq \tau \implies L_0 \cdot \dots \cdot L_{n-1} \cdot \mathtt{O}_{\mathtt{W}}(x) \cdot L_n \in \mathcal{B}(\pi) \land x \notin \mathtt{loc}(L_1 \cdot \dots \cdot L_{n-1}))) \end{array}}{\mathcal{B} \xrightarrow{\tau, \mathtt{W}(x, v_{\mathtt{W}})}_{\text{loLRA}} \mathcal{B}'}$$

READ

$$\frac{\begin{array}{c} \mathtt{loc}(o) = x \\ \mathtt{val}(o) = v_{\mathtt{R}} \\ \mathcal{B} = \mathcal{B}'[\tau \mapsto o \cdot \mathcal{B}'(\tau)] \end{array}}{\mathcal{B} \xrightarrow{\tau, \mathtt{R}(x, v_{\mathtt{R}})}_{\text{loLRA}} \mathcal{B}'}$$

RMW

$$\frac{\begin{array}{c} \mathtt{loc}(o) = x \qquad \mathtt{val}(o) = v_{\mathtt{R}} \qquad \mathtt{rmw\text{-}tid}(o) = \tau \\ \mathcal{B} = \mathcal{B}_{\text{mid}}[\tau \mapsto o \cdot \mathcal{B}_{\text{mid}}(\tau)] \\ \mathcal{B}_{\text{mid}} \xrightarrow{\tau, \mathtt{W}(x, v_{\mathtt{W}})}_{\text{loLRA}} \mathcal{B}' \end{array}}{\mathcal{B} \xrightarrow{\tau, \mathtt{RMW}(x, v_{\mathtt{R}}, v_{\mathtt{W}})}_{\text{loLRA}} \mathcal{B}'}$$

LOWER

$$\frac{\mathcal{B}' \sqsubseteq \mathcal{B}}{\mathcal{B} \xrightarrow{\varepsilon}_{\text{loLRA}} \mathcal{B}'}$$

Fig. 3. Transitions of loLRA memory system

- WRITE: For a thread τ to write v to x, an option $\mathtt{O}_{\mathtt{W}}(x)$ must be the first in each of τ's lists. The WRITE consumes these options, discarding the first element from each of τ's lists. To allow future reads from the executed write, the write may add a read option o with $\mathtt{loc}(o) = x$, $\mathtt{val}(o) = v$, $\mathtt{tid}(o) = \tau$, and some $\mathtt{rmw\text{-}tid}(o)$ (possibly multiple times) in every existing list of every thread (including the writer itself). The WRITE step enforces carefully tailored conditions on *where* these new options are added:
  1. In the potential of the writer itself, a new option cannot be added after an existing write option to x (except for the write option that is consumed in this write step) and the last added read option should immediately precede an existing write option to x.
  2. In the potential of other threads the last added read option should immediately precede an existing write option to x that is to be consumed by the current write step.
  3. If more than one option is added, the added read options can never "surround" an existing read/write option with location x.
  4. New read options can be placed in a list L only if the suffix of L after the first occurrence of the newly added read options are present as an option list of the writing thread τ.
- RMW: The only additional requirement when performing an RMW compared to a non-interrupted execution of a read followed by a write is that two RMWs should never read from the same event. This is achieved by including *RMW thread identifiers* in read options, denoting the (unique) thread that may consume this option when executing an RMW. When a thread writes, it picks an (arbitrary) unique thread identifier ($\pi_{\mathtt{RMW}}$) for its added options; reads ignore this field; and RMWs by thread τ can only consume read options whose RMW thread identifier is τ.

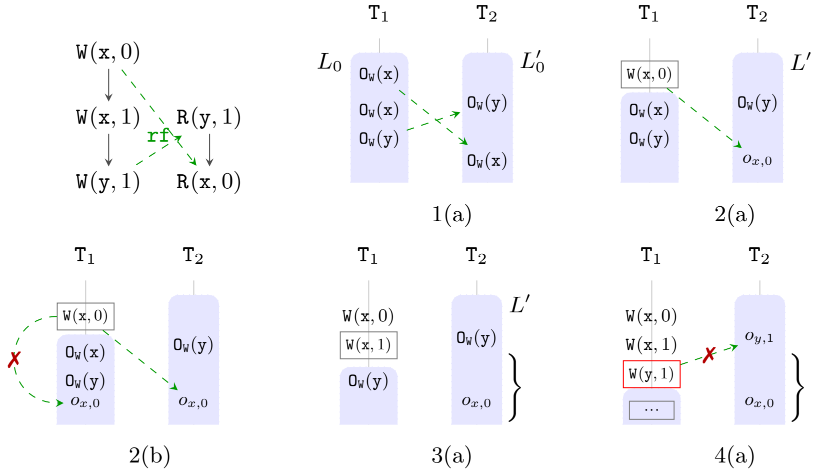


Fig. 4. This figure shows the loLRA transitions for MP program. Here the dashed line in 1(a) between $\mathtt{O_W}(x)$ of $\mathtt{T}_1$ and $\mathtt{O_W}(x)$ of $\mathtt{T}_2$ indicates that a future write $\mathtt{W}(x, 0)$ of $\mathtt{T}_1$ (see 2(a)) may replace the $\mathtt{O_W}(x)$ of $\mathtt{T}_2$ with a read option $o_{x,0}$. We follow a similar depiction in all the remaining diagrams of the paper.

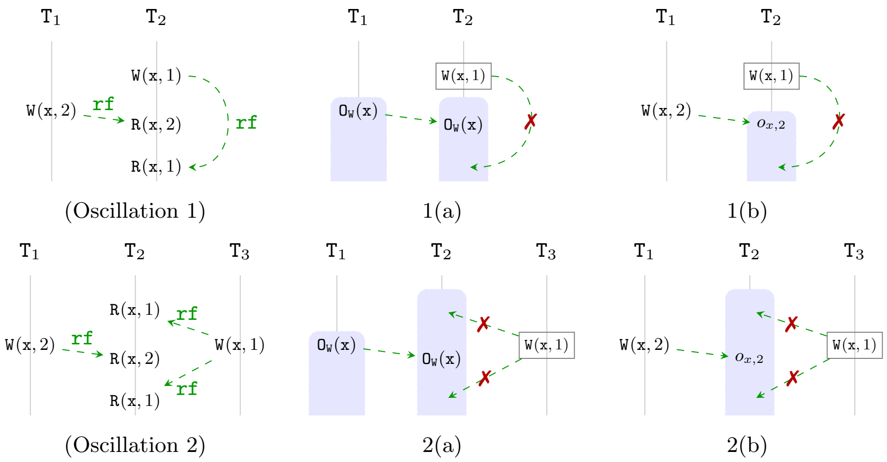
- LOWER: The step allows to remove read/write options as well as full option lists at any time.

We revisit the examples from §3 to illustrate that loLRA forbids those outcomes. In following discussions, shaded portions of the diagram for each thread correspond to its option lists. We write $o_{x,v}$ to represent a read option o with $\mathtt{loc}(o) = x$ and $\mathtt{val}(o) = v$.

Example 2. Recall the execution graph of MP from §3 (see Figure 4). Since no step in loLRA can introduce a write option, we observe the following facts about the option lists $L_0 \in \mathcal{B}_0(\mathtt{T}_1)$ and $L'_0 \in \mathcal{B}_0(\mathtt{T}_2)$ where $\mathcal{B}_0$ may lead to the annotated program state ($\mathtt{a} = 1$ and $\mathtt{b} = 0$) using a trace in which L_0 and L'_0 are not discarded by a LOWER step:

1. L_0 contains $\mathtt{O_W(x)} \cdot \mathtt{O_W(x)} \cdot \mathtt{O_W(y)}$ as a sub-list to enable $\mathtt{W(x, 0)}$, $\mathtt{W(x, 1)}$, and $\mathtt{W(y, 1)}$ in $\mathtt{T}_1$.
2. For the reads $\mathtt{R(y, 1)}$ and $\mathtt{R(x, 0)}$ to happen the corresponding writes $\mathtt{W(y, 1)}$ and $\mathtt{W(x, 0)}$ need to insert read options $o_{\mathrm{y},1}$ and $o_{\mathrm{x},0}$ at these locations (see READ step).
3. L'_0 contains $\mathtt{O_W(y)}$ followed by $\mathtt{O_W(x)}$ to enable future insertions of read options $o_{\mathrm{y},1}$ and $o_{\mathrm{x},0}$ by the writes $\mathtt{W(y, 1)}$ and $\mathtt{W(x, 0)}$ respectively (see condition 2 of WRITE step).

Starting in the state $\mathcal{B}_0$ (1(a) in Figure 4), one can reach state 3(a) through state 2(a) in two successive steps corresponding to execution of the first two writes, $\mathtt{W(x, 0)}$ and $\mathtt{W(x, 1)}$ of $\mathtt{T}_1$, where the first write $\mathtt{W(x, 0)}$ replaces $\mathtt{O_W(x)}$ in the option list of $\mathtt{T}_2$ with a read option $o_{\mathrm{x},0}$ resulting in $L' = L'_0[\mathtt{O_W(x)} \mapsto o_{\mathrm{x},0}]$.



(Oscillation 1) 1(a) 1(b)

(Oscillation 2) 2(a) 2(b)

Fig. 5. loLRA transitions for Oscillation 1 and Oscillation 2 (Example 3).

In the next step (shown as 4(a)), we hope to perform the write $\mathtt{W}(\mathtt{y}, 1)$ in $\mathtt{T}_1$ and replace $\mathtt{O}_{\mathtt{W}}(\mathtt{y})$ in $\mathtt{T}_2$ with the read option $o_{\mathtt{y},1}$. However, the current write step requires that the suffix of L' after $\mathtt{O}_{\mathtt{W}}(\mathtt{y})$ (here, $o_{\mathtt{x},0}$) be present as an option list of thread $\mathtt{T}_1$ (due to condition 4 of the WRITE step). This is clearly not true and hence we can not continue with the current execution trace. To circumvent this blocking run the first write $\mathtt{W}(\mathtt{x}, 0)$ of $\mathtt{T}_1$ might want to non-deterministically insert a read option $o_{\mathtt{x},0}$ at the specified location (see 2(b)) in its option list. However, due to the presence of an earlier $\mathtt{O}_{\mathtt{W}}(\mathtt{x})$ in the option lists of $\mathtt{T}_1$ this is not allowed. Therefore, the loLRA semantics successfully forbids the annotated outcome of the message passing test.

Example 3. Recall the execution graphs of (Oscillation 1) and (Oscillation 2) from §3 (see Figure 5), where $\mathtt{T}_2$ oscillates between the observed values of $\mathtt{x}$. Consider following two cases (and the corresponding execution graphs) to observe a contradiction for each possible trace of loLRA:

- $\mathtt{W}(\mathtt{x}, 1)$ executes before $\mathtt{W}(\mathtt{x}, 2)$: For (Oscillation 1) and (Oscillation 2) this is depicted as 1(a) and 2(a) of Figure 5 respectively. Note the presence of $\mathtt{O}_{\mathtt{W}}(\mathtt{x})$ at the specified locations in the option lists of thread $\mathtt{T}_2$ to mark the end of new read options due to the future write $\mathtt{W}(\mathtt{x}, 2)$. In the current state of (Oscillation 1), the write $\mathtt{W}(\mathtt{x}, 1)$ of thread $\mathtt{T}_2$ is not allowed to put a read option in its own option list due to the presence of an earlier $\mathtt{O}_{\mathtt{W}}(\mathtt{x})$ (see condition 1 of WRITE step). Similarly in the current state of (Oscillation 2), the write $\mathtt{W}(\mathtt{x}, 1)$ of thread $\mathtt{T}_3$ cannot place new read options in the list of thread $\mathtt{T}_2$ because $\mathtt{O}_{\mathtt{W}}(\mathtt{x})$ appears between the new read options (see condition 3 of WRITE step).
- $\mathtt{W}(\mathtt{x}, 1)$ executes after $\mathtt{W}(\mathtt{x}, 2)$: For (Oscillation 1) and (Oscillation 2) this is depicted as 1(b) and 2(b) of Figure 5 respectively. Note the presence of $o_{\mathtt{x},2}$ (instead of $\mathtt{O}_{\mathtt{W}}(\mathtt{x})$ in the previous case) at the specified location in the option

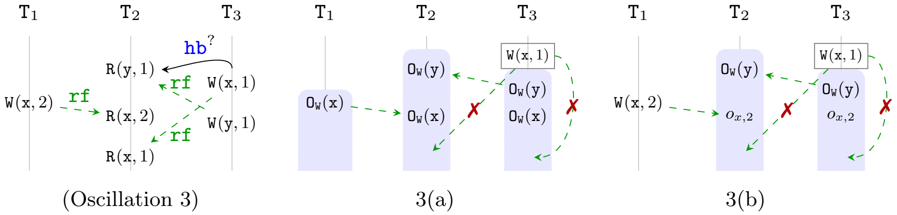


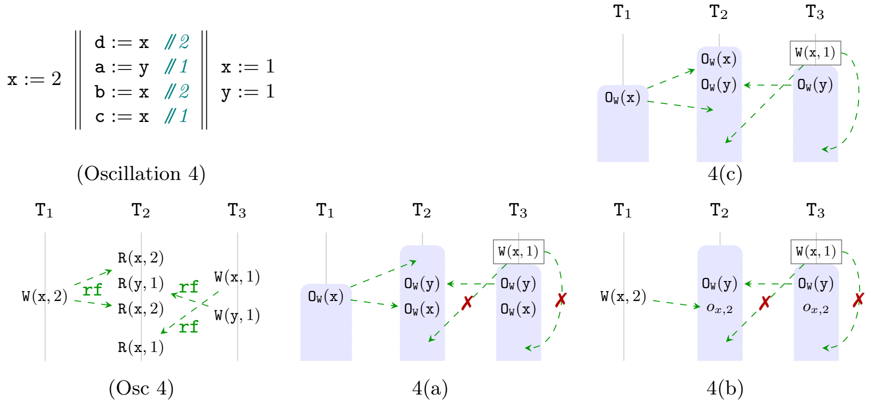
Fig. 6. The loLRA transitions for the program Oscillation 3 (Example 4).

lists of thread $\mathtt{T}_2$ to allow the read $\mathtt{R}(\mathtt{x}, 2)$ to read in future from the write $\mathtt{W}(\mathtt{x}, 2)$ of $\mathtt{T}_1$. Again in the states corresponding to 1(b) and 2(b), due to conditions 1 and 3 of WRITE step, $\mathtt{W}(\mathtt{x}, 1)$ is not allowed to put new read options at the specified locations.

Example 4. Recall the execution graph of (Oscillation 3) from §3, where $\mathtt{T}_2$ oscillates between the observed values of $\mathtt{x}$ (see Figure 6). We consider the following two cases and the resulting execution graphs, based on the order of execution between the write events $\mathtt{W}(\mathtt{x}, 1)$ and $\mathtt{W}(\mathtt{x}, 2)$, to observe a contradiction in each trace of loLRA:

– $\mathtt{W}(\mathtt{x}, 1)$ executes before $\mathtt{W}(\mathtt{x}, 2)$: This condition is depicted as 3(a). Note the presence of $\mathtt{O}_\mathtt{W}(\mathtt{y})$ and $\mathtt{O}_\mathtt{W}(\mathtt{x})$ at the specified location in the option list of $\mathtt{T}_2$ to mark the end of new read options due to the future writes $\mathtt{W}(\mathtt{y}, 1)$ and $\mathtt{W}(\mathtt{x}, 2)$ of $\mathtt{T}_3$ and $\mathtt{T}_1$ respectively. Also note the presence of $\mathtt{O}_\mathtt{W}(\mathtt{x})$ in the option lists of $\mathtt{T}_3$. We claim that this $\mathtt{O}_\mathtt{W}(\mathtt{x})$ is needed as justification for the future write $\mathtt{W}(\mathtt{y}, 1)$ of $\mathtt{T}_3$ (when the write $\mathtt{W}(\mathtt{y}, 1)$ will be replacing the write option $\mathtt{O}_\mathtt{W}(\mathtt{y})$ on $\mathtt{T}_2$ with the read option $o_{\mathrm{y},1}$). To justify the claim, assume otherwise (i.e., $\mathtt{O}_\mathtt{W}(\mathtt{x})$ is absent in the option list of $\mathtt{T}_3$), and we observe that $\mathtt{W}(\mathtt{y}, 1)$ of $\mathtt{T}_3$ can not continue in any of the following possible cases:
  - $\mathtt{W}(\mathtt{x}, 2)$ has not occurred when $\mathtt{W}(\mathtt{y}, 1)$ tries to execute: In this case $\mathtt{O}_\mathtt{W}(\mathtt{x})$ is still present in the option list of $\mathtt{T}_2$ and hence is required at the specified location in the option list of $\mathtt{T}_3$ as a justification for the current write $\mathtt{W}(\mathtt{y}, 1)$ (see condition 4 of the WRITE step). Therefore, the write $\mathtt{W}(\mathtt{y}, 1)$ can not continue in this case.
  - $\mathtt{W}(\mathtt{x}, 2)$ has occurred when $\mathtt{W}(\mathtt{y}, 1)$ tries to execute: In this case $\mathtt{O}_\mathtt{W}(\mathtt{x})$ on $\mathtt{T}_2$ has been replaced with a $o_{\mathrm{x},2}$ and hence $o_{\mathrm{x},2}$ is also expected in the option list of $\mathtt{T}_3$ (as justification for the current WRITE step $\mathtt{W}(\mathtt{y}, 1)$). However, the presence of $o_{\mathrm{x},2}$ in $\mathtt{T}_3$ can only be ensured (as insertion of new read option) by the corresponding write $\mathtt{W}(\mathtt{x}, 2)$. The write $\mathtt{W}(\mathtt{x}, 2)$ can not add a $o_{\mathrm{x},2}$ at the specified location due to the absence of $\mathtt{O}_\mathtt{W}(\mathtt{x})$ at the same location to mark the end of newly added read options (see condition 2 of WRITE step). Hence, in this case again the write $\mathtt{W}(\mathtt{y}, 1)$ can not continue.

  Assuming the presence of $\mathtt{O}_\mathtt{W}(\mathtt{x})$ in the option list of $\mathtt{T}_3$ (as shown in 3(a)) it is easy to see that $\mathtt{W}(\mathtt{x}, 1)$ of $\mathtt{T}_3$ can not put a read option in its own option



```
x := 2 ‖ d := x  // 2  ‖ x := 1
       ‖ a := y  // 1  ‖ y := 1
       ‖ b := x  // 2  ‖
       ‖ c := x  // 1  ‖
```

(Oscillation 4)

(Osc 4) 4(a) 4(b) 4(c)

Fig. 7. The loLRA transitions for the program Oscillation 4 (Example 5).

list (see condition 1 of WRITE step) which is necessary as justification for the future write W(y, 1) of $\mathtt{T}_3$ (again using similar arguments as discussed above). Therefore, the current case is forbidden by the lossy loLRA semantics.

- W(x, 1) executes after W(x, 2): This condition is depicted as 3(b), where the write W(x, 2) of $\mathtt{T}_1$ has replaced the write option $\mathtt{O}_\mathtt{W}(\mathtt{x})$ in the option lists of $\mathtt{T}_2$ with a read option $o_{\mathtt{x},2}$. Again, as discussed in the previous case (for justifying the future write W(y, 1) of $\mathtt{T}_3$), the write W(x, 2) of $\mathtt{T}_1$ should also place a read option $o_{\mathtt{x},2}$ in the option lists of $\mathtt{T}_3$ at the specified location. Now, as shown in 3(b), the write W(x, 1) of $\mathtt{T}_3$ can not put a read option in its own option list (due to the presence of an earlier $o_{\mathtt{x},2}$) which is necessary as justification for the future write W(y, 1) of $\mathtt{T}_3$. Thus, the current case is also forbidden by the lossy loLRA semantics.

In the discussions so far (particularly related to cases 1(a), 2(a), and 3(a) of the previous examples), we observed that marking the end of newly added read options (using a pre-existing write option) is helpful in forbidding oscillations. In all of these cases it is easy to see (using exactly similar arguments) that we can also forbid these oscillatory behaviors by requiring (in conditions 1 and 2 of the WRITE step) that the beginning of newly added read options be marked using a pre-existing write option. Next example illustrates the distinctive advantage of marking the end over marking the beginning.

Example 5. Consider execution graph (Osc 4) corresponding to the annotated outcome of (Oscillation 4) shown in Figure 7. The constraint local-read-coherence forbids this execution graph since we have (1) $G.\mathtt{hb}|_{\mathtt{loc}} \setminus G.\mathtt{rf}$ from W(x, 1) to the third read R(x, 2) of $\mathtt{T}_2$; (2) $G.\mathtt{hb}$ from the third read R(x, 2) of $\mathtt{T}_2$ to the last read R(x, 1) of $\mathtt{T}_2$; and (3) rf from W(x, 1) to the last read R(x, 1) of $\mathtt{T}_2$.

Consider the following two possibilities (4(b) and 4(a) of Figure 7) corresponding to this outcome where: (1) W(x, 1) executes after W(x, 2); and (2) W(x, 1) executes before W(x, 2).

Assuming (1) and using arguments similar to Example 4, we land in configuration 4(b) which is not allowed by the lossy loLRA semantics. However, note that assuming (2) we get a contradiction only because $\mathtt{O}_{\mathtt{W}}(\mathtt{x})$ is present at the specified location in 4(a) to mark the end of new read options in the option list of $\mathtt{T}_2$ (by the write $\mathtt{W}(\mathtt{x}, 2)$ of thread $\mathtt{T}_1$). Instead, if we choose to mark the beginning (and not the end) of new read options in the option list of $\mathtt{T}_2$ we result in the configuration of 4(c) resulting in the absence of any pre-existing $\mathtt{O}_{\mathtt{W}}(\mathtt{x})$ at the end of the new entries. In this case, we observe that there is a trace of lossy loLRA (for the annotated outcome of (Oscillation 4)) in which $\mathtt{W}(\mathtt{x}, 1)$ and $\mathtt{W}(\mathtt{y}, 1)$ of $\mathtt{T}_3$ appears before $\mathtt{W}(\mathtt{x}, 2)$ of $\mathtt{T}_1$.

Next, we show that for a given program Pr, $Pr \bowtie$loLRA admits the required conditions of the WSTS framework that ensure decidability of the induced coverability problem (see, e.g., [9, 15]). In particular, the compatibility condition between the well-quasi-ordering on states and the transitions is trivial since we explicitly include the (LOWER) step in loLRA.

Lemma 1. *Given a program Pr, the LTS $Pr \bowtie lo$LRA equipped with the well-quasi-ordering $\sqsubseteq$ (lifted to states of $Pr \bowtie lo$LRA by defining $\langle \overline{p}, \mathcal{B} \rangle \sqsubseteq \langle \overline{p}', \mathcal{B}' \rangle$ iff $\overline{p} = \overline{p}'$ and $\mathcal{B} \sqsubseteq \mathcal{B}'$) is a WSTS that admits effective initialization and effective pred-basis.*

As a corollary, we obtain that state reachability under loLRA is decidable. We refer the reader to [32] where we give more details and proofs (which generally follow those in [22]).

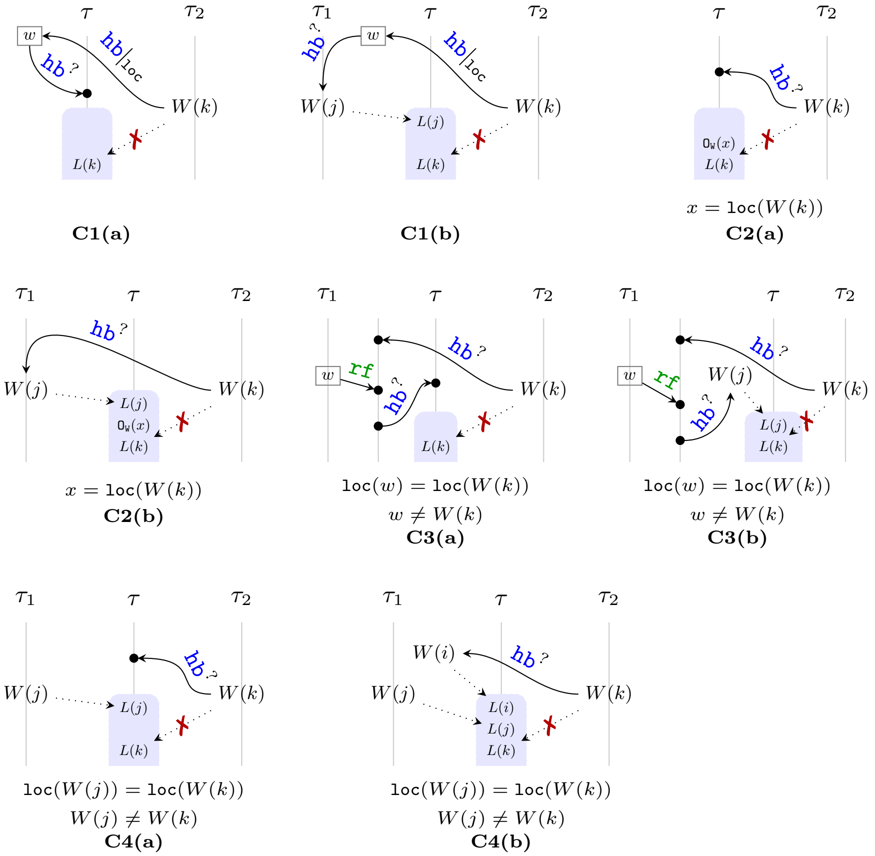
5 Equivalence of the Memory Systems for LRA

In this section we establish the equivalence between loLRA and opLRA by demonstrating a simulation between these systems. The states of loLRA and opLRA are related to each other using *write lists*, which match read options in loLRA's potentials with concrete write event in opLRA's execution graphs.

Definition 6. A *write list* is a sequence of write events and write options. Let G be an execution graph, L an option list, and $tid_{\mathtt{RMW}} : \mathsf{W} \rightarrow \mathsf{Tid}$. A write list W is a $\langle G, L, tid_{\mathtt{RMW}} \rangle$*-write-list* if $|L| = |W|$ and the following hold for every $1 \leq k \leq |W|$:

- If $L(k)$ is a write option, then $W(k) = L(k)$.
- If $L(k) = \langle \tau, x, v, \pi_{\mathtt{RMW}} \rangle$, then $W(k) \in G.\mathsf{W}$, $\mathtt{tid}(W(k)) = \tau$, $\mathtt{loc}(W(k)) = x$, $\mathtt{val}_{\mathtt{W}}(W(k)) = v$, and $tid_{\mathtt{RMW}}(W(k)) = \pi_{\mathtt{RMW}}$.

In addition to the above, we require that weak-coherence and local-read-coherence are maintained by any extension of the execution graph G with a sequence of reads and writes of thread τ that are obtained by following the write list W. This is formalized in the following notion of $\langle G, \tau \rangle$*-consistency* of a write list W.



C1(a)

C1(b)

$x = \mathtt{loc}(W(k))$
C2(a)

$x = \mathtt{loc}(W(k))$
C2(b)

$\mathtt{loc}(w) = \mathtt{loc}(W(k))$
$w \neq W(k)$
C3(a)

$\mathtt{loc}(w) = \mathtt{loc}(W(k))$
$w \neq W(k)$
C3(b)

$\mathtt{loc}(W(j)) = \mathtt{loc}(W(k))$
$W(j) \neq W(k)$
C4(a)

$\mathtt{loc}(W(j)) = \mathtt{loc}(W(k))$
$W(j) \neq W(k)$
C4(b)

Fig. 8. Illustration of conditions in Definition 7 for the $\langle G, \tau\rangle$-*consistency* of W. Each condition is split into two cases (e.g., C1 is summarized using C1(a) or C1(b)).

Definition 7. A write list W is $\langle G, \tau\rangle$-*consistent* if for every $1 \le k \le |W|$ with $W(k) \in \mathsf{E}$:

C1 $W(k) \notin dom(G.\mathtt{hb}|_{\mathtt{loc}} ; [\mathsf{W}] ; G.\mathtt{hb}^? ; [\mathsf{E}^\tau \cup \{W(j) \mid 1 \le j < k\}])$.
C2 If $W(i) = \mathtt{O_W}(\mathtt{loc}(W(k)))$ for some $i < k$,
then $W(k) \notin dom(G.\mathtt{hb}^? ; [\mathsf{E}^\tau \cup \{W(j) \mid 1 \le j < i\}])$.
C3 $W(k) \notin dom((G.\mathtt{hb}|_{\mathtt{loc}} \setminus G.\mathtt{rf}) ; [\mathsf{R}] ; G.\mathtt{hb}^? ; [\mathsf{E}^\tau \cup \{W(j) \mid 1 \le j < k\}])$.
C4 If $\mathtt{loc}(W(j)) = \mathtt{loc}(W(k))$ and $W(k) \neq W(j)$ for some $j < k$,
then $W(k) \notin dom(G.\mathtt{hb}^? ; [\mathsf{E}^\tau \cup \{W(i) \mid 1 \le i < j\}])$.

Intuitively, for any future extension of execution graph with a sequence of events on τ, conditions C1 and C2 help in maintaining weak-coherence while C3 and C4 ensure that local-read-coherence is preserved. To assist readers, these conditions are depicted using diagrams in Figure 8 where the shaded area of τ represents a sequence of future events.

The simulation relation Υ is now defined as follows.

Definition 8. A state $\mathcal{B} \in$ loLRA.$\mathtt{Q}$ *matches* an execution graph G, denoted by $\mathcal{B} \curlyvee G$, if there exists a function $\mathit{tid}_{\mathtt{RMW}} : \mathsf{W} \to \mathsf{Tid}$, such that: (1) for every $\tau \in \mathsf{Tid}$ and $L \in \mathcal{B}(\tau)$, there exists a $\langle G, \tau \rangle$-consistent $\langle G, L, \mathit{tid}_{\mathtt{RMW}} \rangle$-write-list, and (2) for every $\langle w, e \rangle \in G.\mathtt{rf}\,;[\mathsf{RMW}]$, we have $\mathtt{tid}(e) = \mathit{tid}_{\mathtt{RMW}}(w)$.

Based on the simulation relation, we establish the equivalence of loLRA and opLRA. The proof, given in [32], shows that $\curlyvee$ constitutes a forward simulation from loLRA to opLRA, and $\curlyvee^{-1}$ constitutes a backward simulation from opLRA to loLRA.

Theorem 1. *The traces of lo*LRA *and the traces of op*LRA *coincide.*

6 Conclusion, Related and Future Work

We established the decidability of state reachability for finite-state programs under LRA, a memory model that lies strictly between WRA and RA. For that matter, we adapted the potential-based semantics of WRA from [22] to LRA, and showed that it meets the requirements for decidability of the WSTS framework.

In addition to the closely related work discussed in the introduction to this paper, the paper [14] studies the problem of verifying whether a given memory system provides causal consistency, which is a different verification problem than the one discussed in the current paper. The CC model in [14] (when restricted to single instruction transactions) is equivalent to (the RMW-free fragment of) WRA, whereas CCv from [14] is equivalent to SRA.

Another line of related work concerns *parametrized* programs, where one has an unknown number of threads but all of them run the same code. This arises a decidable verification problem under SC and TSO [5], but decidability of this problem is still unknown for WRA, SRA, and LRA. For the RMW-free fragment this problem is PSPACE for TSO [8] as well as for RA [19] (the latter result also allows a fixed number of distinguished threads running loop-free programs, possibly including RMWs).

An interesting direction for future work is to try to further close the gap between LRA and RA by introducing a restricted form of RA's modification order. A related problem that is still open (to the best of our knowledge) is whether the fragment of RA without RMWs induces a decidable verification problem. In addition, other models with undecidable reachability problems (such as the promising semantics [6] and the full POWER model [3]) may be bounded from below by decidable models.

Acknowledgements This work was supported by the Israel Science Foundation (grant number 814/22) and the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement no. 851811).

References

1. Abdulla, P.A.: Well (and better) quasi-ordered transition systems. The Bulletin of Symbolic Logic **16**(4), 457–515 (2010), http://www.jstor.org/stable/40961367
2. Abdulla, P.A., Arora, J., Atig, M.F., Krishna, S.: Verification of programs under the release-acquire semantics. In: PLDI. pp. 1117–1132. ACM, New York, NY, USA (2019). https://doi.org/10.1145/3314221.3314649
3. Abdulla, P.A., Atig, M.F., Bouajjani, A., Derevenetc, E., Leonardsson, C., Meyer, R.: On the state reachability problem for concurrent programs under Power. In: NETYS. pp. 47–59. Springer International Publishing, Cham (2021). https://doi.org/10.1007/978-3-030-67087-0_4
4. Abdulla, P.A., Atig, M.F., Bouajjani, A., Ngo, T.P.: The benefits of duality in verifying concurrent programs under TSO. In: CONCUR. LIPIcs, vol. 59, pp. 5:1–5:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2016), https://doi.org/10.4230/LIPIcs.CONCUR.2016.5
5. Abdulla, P.A., Atig, M.F., Bouajjani, A., Ngo, T.P.: 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
6. Abdulla, P.A., Atig, M.F., Godbole, A., Krishna, S., Vafeiadis, V.: The decidability of verification under PS 2.0. In: ESOP. pp. 1–29. Springer International Publishing, Cham (2021). https://doi.org/10.1007/978-3-030-72019-3_1
7. Abdulla, P.A., Atig, M.F., Jonsson, B., Ngo, T.P.: Optimal stateless model checking under the release-acquire semantics. Proc. ACM Program. Lang. **2**(OOPSLA), 135:1–135:29 (Oct 2018). https://doi.org/10.1145/3276505
8. Abdulla, P.A., Atig, M.F., Rezvan, R.: Parameterized verification under TSO is PSPACE-complete. Proc. ACM Program. Lang. **4**(POPL) (Dec 2019). https://doi.org/10.1145/3371094
9. Abdulla, P.A., Čerāns, K., Jonsson, B., Tsay, Y.K.: Algorithmic analysis of programs with well quasi-ordered domains. Information and Computation **160**(1), 109–127 (2000). https://doi.org/10.1006/INCO.1999.2843
10. Ahamad, M., Neiger, G., Burns, J.E., Kohli, P., Hutto, P.W.: Causal memory: definitions, implementation, and programming. Distributed Computing **9**(1), 37–49 (1995). https://doi.org/10.1007/BF01784241
11. Alglave, J., Maranget, L., Tautschnig, M.: Herding cats: modelling, simulation, testing, and data mining for weak memory. ACM Trans. Program. Lang. Syst. **36**(2), 7:1–7:74 (Jul 2014). https://doi.org/10.1145/2627752
12. Atig, M.F., Bouajjani, A., Burckhardt, S., Musuvathi, M.: What's decidable about weak memory models? In: ESOP. pp. 26–46. Springer-Verlag, Berlin, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28869-2_2
13. Batty, M., Owens, S., Sarkar, S., Sewell, P., Weber, T.: Mathematizing C++ concurrency. In: POPL. pp. 55–66. ACM, New York, NY, USA (2011). https://doi.org/10.1145/1925844.1926394
14. Bouajjani, A., Enea, C., Guerraoui, R., Hamza, J.: On verifying causal consistency. In: POPL. pp. 626–638. ACM, New York, NY, USA (2017). https://doi.org/10.1145/3009837.3009888
15. Finkel, A., Schnoebelen, P.: Well-structured transition systems everywhere! Theoretical Computer Science **256**(1), 63 – 92 (2001). https://doi.org/10.1016/S0304-3975(00)00102-X
16. ISO/IEC 14882:2011: Programming language C++ (2011)
17. ISO/IEC 9899:2011: Programming language C (2011)

18. Kaiser, J.O., Dang, H.H., Dreyer, D., Lahav, O., Vafeiadis, V.: Strong logic for weak memory: Reasoning about release-acquire consistency in Iris. In: ECOOP. pp. 17:1–17:29. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2017). https://doi.org/10.4230/LIPIcs.ECOOP.2017.17
19. Krishna, S., Godbole, A., Meyer, R., Chakraborty, S.: Parameterized verification under release acquire is PSPACE-complete. In: PODC. pp. 482–492. ACM, New York, NY, USA (2022). https://doi.org/10.1145/3519270.3538445
20. Lahav, O.: Verification under causally consistent shared memory. ACM SIGLOG News **6**(2), 43–56 (Apr 2019). https://doi.org/10.1145/3326938.3326942
21. Lahav, O., Boker, U.: Decidable verification under a causally consistent shared memory. In: PLDI. pp. 211–226. ACM (2020). https://doi.org/10.1145/3385412.3385966
22. Lahav, O., Boker, U.: What's Decidable About Causally Consistent Shared Memory? ACM Trans. Program. Lang. Syst. **44**(2), 8:1–8:55 (2022), https://doi.org/10.1145/3505273
23. Lahav, O., Giannarakis, N., Vafeiadis, V.: Taming release-acquire consistency. In: POPL. pp. 649–662. ACM, New York, NY, USA (2016). https://doi.org/10.1145/2837614.2837643
24. Lahav, O., Margalit, R.: Robustness against release/acquire semantics. In: PLDI. pp. 126–141. ACM, New York, NY, USA (2019). https://doi.org/10.1145/3314221.3314604
25. Lahav, O., Vafeiadis, V.: Owicki-gries reasoning for weak memory models. In: ICALP. pp. 311–323. Springer-Verlag, Berlin, Heidelberg (2015). https://doi.org/10.1007/978-3-662-47666-6_25
26. Lahav, O., Vafeiadis, V.: Explaining relaxed memory models with program transformations. In: FM. LNCS, vol. 9995, pp. 479–495. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-319-48989-6_29
27. Lahav, O., Vafeiadis, V., Kang, J., Hur, C.K., Dreyer, D.: Repairing sequential consistency in C/C++11. In: PLDI. pp. 618–632. ACM, New York, NY, USA (2017). https://doi.org/10.1145/3062341.3062352
28. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans. Computers **28**(9), 690–691 (1979)
29. Owens, S., Sarkar, S., Sewell, P.: A better x86 memory model: x86-TSO. In: TPHOLs. pp. 391–407. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03359-9_27
30. Raad, A., Lahav, O., Vafeiadis, V.: On parallel snapshot isolation and release/acquire consistency. In: ESOP. pp. 940–967. Springer, Berlin, Heidelberg (2018). https://doi.org/10.1007/978-3-319-89884-1_33
31. Sewell, P., Sarkar, S., Owens, S., Zappa Nardelli, F., Myreen, M.O.: x86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. Commun. ACM **53**(7), 89–97 (2010). https://doi.org/10.1145/1785414.1785443
32. Singh, A.K., Lahav, O.: Decidable verification under localized release-acquire concurrency (extended version) (2024), https://www.cs.tau.ac.il/~orilahav/papers/tacas24full.pdf
33. Vafeiadis, V., Balabonski, T., Chakraborty, S., Morisset, R., Zappa Nardelli, F.: Common compiler optimisations are invalid in the C11 memory model and what we can do about it. In: POPL. pp. 209–220. ACM, New York, NY, USA (2015). https://doi.org/10.1145/2676726.2676995