Robustness against Release/Acquire Semantics

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

Roy Margalit
Tel Aviv University
Israel
roy.margalit@cs.tau.ac.il

Abstract

We present an algorithm for automatically checking robustness of concurrent programs against C/C++11 release/acquire semantics, namely verifying that all program behaviors under release/acquire are allowed by sequential consistency. Our approach reduces robustness verification to a reachability problem under (instrumented) sequential consistency. We have implemented our algorithm in a prototype tool called *Rocker* and applied it to several challenging concurrent algorithms. To the best of our knowledge, this is the first precise method for verifying robustness against a high-level programming language weak memory semantics.

CCS Concepts • **Theory of computation** → *Verification by model checking*; *Concurrent algorithms*; *Program semantics*; *Program verification*; *Program analysis*; • **Software and its engineering** → *Software verification*.

Keywords weak memory models, C/C++11, release/acquire, robustness

ACM Reference Format:
Ori Lahav and Roy Margalit. 2019. Robustness against Release/Acquire Semantics. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3314221.3314604

1 Introduction

Release/acquire (RA), the fragment of the C/C++11 memory model [14] consisting of release stores, acquire loads and acquire-release read-modify-writes (RMWs), is a particularly useful and well-behaved weak memory model [36]. It is weaker than sequential consistency (SC) [40] and allows higher performance implementations. For example, x86-TSO [50] provides RA "for free" (its memory model is stronger than RA), and POWER [45] implements RA using 'lightweight sync' instructions rather than more expensive 'full sync' instructions which are needed for SC.

At the same time, since RA is designed to support the common "message passing" synchronization idiom, the guarantees provided by RA suffice to implement various fundamental concurrent algorithms and synchronization mechanisms. In fact, many useful programs are actually *robust against RA*—the behaviors they exhibit under RA semantics are also allowed under SC—or can be made robust by placing few *SC-fences* or by strengthening certain reads and writes to be RMW operations. Such modifications are sometimes necessary, with the best known example being Dekker's mutual exclusion algorithm, whose RA (non-SC) behavior is harmful for its correctness.

A natural question is thus whether one can automatically verify robustness against RA. Our main contribution is a decision procedure for this problem. Besides our theoretical interest, we believe that this result can facilitate the development of concurrent algorithms for RA. In particular, if we are able to verify robustness against RA, various programs designed for SC may be directly ported and verified with more ordinary techniques assuming SC. Further, robustness of non-robust programs may be enforced (by placing SC-fences or RMW operations), and verifying the robustness of the strengthened program.

To precisely state our result, it is crucial to carefully define what constitutes a behavior of a concurrent program under SC and under RA, which in turn determines what robustness means. Here, it is natural to use operational presentations of SC and RA as memory subsystems, formulated as labeled transition systems (for RA one could use the timestamp machine introduced in [33]). Then, program behaviors correspond to program states that are reachable when linked with each of the memory subsystems. More precisely, thinking about a concurrent program as a labeled transition system (whose states compromise of the values of the thread-local program counters and variables), one may identify SC (RA) program behaviors with the set of states of the program that are reachable in its runs when synchronized with runs of the SC (RA) memory subsystems. This definition of program behavior leads to what is known as *state robustness*, and corresponds to typical safety properties verification using local assertions and global invariants that relate values of local variables and program counters.

Nevertheless, following [24, Thm. 2.12], it is easy to show that verifying state robustness against RA is as hard as the

PLDI '19, June 22–26, 2019, Phoenix, AZ, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6712-7/19/06.
https://doi.org/10.1145/3314221.3314604

general state reachability problem under RA. The latter problem was recently shown to be undecidable [2]. Thus we resort to a more informative definition of a behavior, leading to a stronger notion of robustness. By doing so, we follow works on robustness against hardware models, TSO in particular (e.g., [17, 19]), where state robustness—like state reachability—is non-primitive recursive [11, 12]. For this matter, we use formulations of SC and RA as labeled transition systems whose states are (C/C++11-like) *execution graphs*. Execution graphs keep track of the full partially ordered history of the run (and thus in this presentation both SC and RA are infinite state systems), including the reads-from mapping (mapping each read to the write it read from) and the modification order (a total order on writes to the same location). The difference between SC and RA is then reduced to the transitions they allow. For instance, when adding reads to the execution graph, SC requires that it reads from the write that is maximal in the modification order, while RA places much weaker restrictions. Now, we can identify program behaviors with pairs of states of both the program and the memory subsystem that are reachable in their synchronized runs. We refer to the robustness notion induced by this definition as *execution-graph robustness*.

Our main contribution is a decision procedure that checks whether a given concurrent program is execution-graph robust against RA. To achieve this, we show how this verification problem can be reduced to a state reachability problem under a (finite state) instrumented SC memory. Roughly speaking, this memory keeps track of the relevant parts of the generated execution graph and uses this information for monitoring that RA execution graphs cannot diverge from SC ones. We prove that our approach is sound and precise. In particular, it follows that this verification problem for programs with bounded data domain is PSPACE-complete.

Our approach can be straightforwardly extended to handle C/C++11's *non-atomic accesses*. A data-race on a non-atomic access is considered an undefined behavior, and, thus, robustness of a program should also imply that it has no data-races on non-atomic accesses. Since robust programs have only SC executions, checking for data-races can be done using standard techniques. For completeness, we incorporated these checks in our method simultaneously to the verification of robustness against RA.

We have implemented our method in a prototype tool, called *Rocker*, using Spin [31] as a back-end model checker under SC. We used *Rocker* to verify the robustness of several concurrent algorithms, including Peterson's mutual execution adaptations for RA [57], sequence locks [16] and user-mode read-copy-update (RCU) implementations [26]. In particular, we observe that execution-graph robustness is a useful property, allowing one, in many cases, to think in terms of SC while running on a weaker model.

The rest of this paper is structured as follows. In §2 we formally present the programming language and the notion of state robustness. In §3 we present the RA concurrency semantics. In §4 we define execution-graph robustness against RA. In §5 we present our decision procedure. In §6 we extend the decision procedure to support non-atomic accesses. In §7 we discuss the implementation and our experiments with it. In §8 we discuss related work. Finally, in §9 we conclude and outline directions for future work. Additional material and proofs for the claims of this paper are available in [1]. The prototype implementation and the examples it was tested on are available in the artifact accompanying this paper.

2 Preliminaries: State Robustness

Given a (binary) relation R, $dom(R)$ and $codom(R)$ denote its domain and codomain, and $R^?$, R^+, and R^* denote its reflexive, transitive, and reflexive-transitive closures. The inverse of a relation R is denoted by R^{-1}, and the (left) composition of two relations R_1, R_2 is denoted by $R_1 \,;\, R_2$. We denote by $[A]$ the identity relation on a set A. In particular, $[A] \,;\, R \,;\, [B] = R \cap (A \times B)$. For a strict total order R, we write $R|_{\text{imm}}$ to denote the set of *immediate R-edges*, i.e., $R|_{\text{imm}} = R \setminus (R \,;\, R)$.

2.1 Programming Language

Let Val, Loc, Reg be finite sets of values, (shared) locations, and register names. We assume that Val contains a distinguished value 0, used as the initial value for all locations. Figure 1 presents our toy programming 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); $\texttt{if}\ e\ \texttt{goto}\ n$ jumps to line n of the program 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; $r := \texttt{FADD}(x, e)$ atomically increments x by the value of e and loads the old value of x to r; and $r := \texttt{CAS}(x, e_{\texttt{R}} \to e_{\texttt{W}})$ atomically loads the value of x to r, compares it to the value of $e_{\texttt{R}}$, and if the two values are equal, replaces the value of x by the value of $e_{\texttt{W}}$.

The less standard instructions wait and BCAS are blocking: $\texttt{wait}(x = e)$ blocks the current thread until it manages to load the value of e from x; and $\texttt{BCAS}(x, e_{\texttt{R}} \to e_{\texttt{W}})$ blocks the current thread until it performs a successful CAS of x from the value of $e_{\texttt{R}}$ (to the value of $e_{\texttt{W}}$). These instructions can be easily implemented using loops (e.g., $L : r := x; \texttt{if}\ r \neq e\ \texttt{goto}\ L$ with fresh r for $\texttt{wait}(x = e)$). Nevertheless, as we demonstrate in the end of this section, including them as primitives leads to a more expressive notion of robustness.

In turn, a sequential program $S \in$ SProg is a finite map from $\mathbb{N}$ to instructions (we assume that $0 \in dom(S)$), and a concurrent program P is a top-level parallel composition of sequential programs, defined as a mapping from a finite set Tid $\subseteq \mathbb{N}$ of thread identifiers to SProg. In our examples, we often write sequential programs as sequences of instructions

$v \in \mathsf{Val}$	Values
$x \in \mathsf{Loc}$	Locations
$r \in \mathsf{Reg}$	Registers
$\tau \in \mathsf{Tid} \subseteq \mathbb{N}$	Thread identifiers

$$\begin{array}{l}\mathsf{Exp} \ni e ::= r \mid v \mid e + e \mid e = e \mid e \neq e \mid \ldots \\ \mathsf{Inst} \ni \mathit{inst} ::= r := e \mid \mathtt{if}\ e\ \mathtt{goto}\ n \mid x := e \mid r := x \mid \\ \qquad r := \mathtt{FADD}(x, e) \mid r := \mathtt{CAS}(x, e \to e) \\ \qquad \mathtt{wait}(x = e) \mid \mathtt{BCAS}(x, e \to e)\end{array}$$

Sequential programs:
$S \in \mathsf{SProg} \triangleq \mathbb{N} \xrightarrow{\mathsf{fin}} \mathsf{Inst}$

Concurrent programs:
$P : \mathsf{Tid} \to \mathsf{SProg}$

Figure 1. Domains and programming language syntax.

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

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

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

delimited by line breaks, use '‖' for parallel composition, and refer to the program threads as $\tau_1, \tau_2, \ldots$ following their left-to-right order in the program listing.

2.2 From Programs to Transition Systems

A *labeled transition system* (LTS) A over an alphabet Σ is a tuple $\langle Q, q_0, \to\rangle$, where Q is a set of *states*, $q_0 \in Q$ is the *initial state*, and $\to \subseteq Q \times \Sigma \times Q$ is a set of *transitions*. We write $\xrightarrow{\sigma}$ for the relation $\{\langle q, q'\rangle \mid \langle q, \sigma, q'\rangle \in \to\}$, and $\to$ for $\bigcup_{\sigma \in \Sigma} \xrightarrow{\sigma}$. We denote by $A.\mathsf{Q}$, $A.\mathsf{q}_0$ and $\to_A$ the three components of an LTS A. A state $q \in A.\mathsf{Q}$ is called *reachable* in A if $A.\mathsf{q}_0 \to_A^* q$. A symbol $\sigma \in \Sigma$ is *enabled* in q (alternatively, q *enables* σ) if $q \xrightarrow{\sigma}_A q'$ for some q'. A sequence $\sigma_1, \ldots, \sigma_n$ is a *trace* of A if $A.\mathsf{q}_0 \xrightarrow{\sigma_1}_A \cdots \xrightarrow{\sigma_n}_A q$ for some q.

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

A sequential program $S \in \mathsf{SProg}$ induces an LTS over $\mathsf{Lab} \cup \{\epsilon\}$, whose states are pairs $\langle pc, \Phi\rangle$ where $pc \in \mathbb{N}$ (called *program counter*) and $\Phi : \mathsf{Reg} \to \mathsf{Val}$ (called *store*, and extended to expressions in the obvious way). Its initial state is $\langle 0, \lambda r.\, 0\rangle$, and its transitions are given in Fig. 2, following the informal description above of the language constructs. In the sequel we identify sequential programs with their induced LTSs (when writing, e.g., $S.\mathsf{Q}$ and $\to_S$).

Example 2.2. We present the LTS induced by a simple sequential program S. Let $\mathsf{Val} = \{0, \ldots, 4\}$, $\mathsf{Loc} = \{x\}$ and $\mathsf{Reg} = \{r\}$. We use $+$ to denote the possibly overflowing sum (e.g., $2 + 4 = 1$), and evaluate expressions of the form $r < e$ to be 1 if $\Phi(r) < \Phi(e)$ and 0 otherwise.

$0 : r := r + 1$
$1 : \mathtt{if}\ r < 2\ \mathtt{goto}\ 0$
$2 : x := r$

$S.\mathsf{Q} = \{0, 1, 2, 3\} \times \{[r \mapsto v] \mid v \in \mathsf{Val}\}$
$S.\mathsf{q}_0 = \langle 0, [r \mapsto 0]\rangle$

$\to_S$ is given by:

$$\begin{array}{l}\{\langle 0, [r \mapsto v]\rangle \xrightarrow{\epsilon}_S \langle 1, [r \mapsto v+1]\rangle \mid v \in \mathsf{Val}\} \cup \\ \{\langle 1, [r \mapsto v]\rangle \xrightarrow{\epsilon}_S \langle 0, [r \mapsto v]\rangle \mid v < 2\} \cup \\ \{\langle 1, [r \mapsto v]\rangle \xrightarrow{\epsilon}_S \langle 2, [r \mapsto v]\rangle \mid v \geq 2\} \cup \\ \{\langle 2, [r \mapsto v]\rangle \xrightarrow{\mathtt{W}(x,v)}_S \langle 3, [r \mapsto v]\rangle \mid v \in \mathsf{Val}\}\end{array}$$

A concurrent program P induces an LTS over the alphabet $\mathsf{Tid} \times (\mathsf{Lab} \cup \{\epsilon\})$. Its states are tuples in $\prod_{\tau \in \mathsf{Tid}} P(\tau).\mathsf{Q}$; its initial state is $\lambda\tau.\, P(\tau).\mathsf{q}_0$; and its transitions are interleaved transitions of P's components, given by:

$$\frac{q_\tau \xrightarrow{l_\epsilon}_{P(\tau)} q'_\tau \qquad \forall \pi \neq \tau.\, q_\pi = q'_\pi}{\lambda\pi.q_\pi \xrightarrow{\langle \tau, l_\epsilon\rangle} \lambda\pi.q'_\pi}$$

In the sequel we identify concurrent programs with their induced LTSs. We often use vector notation (e.g., $\overline{q}$) to denote states of concurrent programs.

2.3 Concurrent Systems and State Robustness

To give semantics to concurrent programs, we synchronize them with *memory subsystems*, as defined next.

Definition 2.3. A *memory subsystem* is a (possibly infinite) LTS over the alphabet $\mathbb{N} \times \mathsf{Lab}$.

The labels here are pairs in $\mathbb{N} \times \mathsf{Lab}$ representing the thread identifier and the label of the performed operation.[1]

[1] This formulation suffices for the purposes of this paper. In a broader context, memory subsystems may also employ internal memory actions, such as propagation from local stores to the main memory in TSO. Extending the definitions to a more general notion of robustness is straightforward.

The most well-known memory subsystem is the one of sequential consistency, denoted here by SC. This memory subsystem simply tracks the most recent value written to each location. Formally, it is defined by $\mathsf{SC}.\mathtt{Q} \triangleq \mathsf{Loc} \to \mathsf{Val}$, $\mathsf{SC}.\mathtt{q}_{\emptyset} \triangleq \lambda x.\, 0$, and $\to_{\mathsf{SC}}$ is given by:

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

Note that SC is oblivious to the thread that takes the action (we have $M \xrightarrow{\langle \tau, l \rangle}_{\mathsf{SC}} M'$ iff $M \xrightarrow{\langle \pi, l \rangle}_{\mathsf{SC}} M'$).

By synchronizing a concurrent program and a memory subsystem, we obtain a *concurrent system* as defined next.

Definition 2.4. A *concurrent system* is a pair, denoted $P_{\mathcal{M}}$, where P is a concurrent program and $\mathcal{M}$ is a memory subsystem. A concurrent system $P_{\mathcal{M}}$ induces an LTS over $\mathsf{Tid} \times \mathsf{Lab}$ whose states are pairs in $P.\mathtt{Q} \times \mathcal{M}.\mathtt{Q}$; its initial state is $\langle P.\mathtt{q}_{\emptyset}, \mathcal{M}.\mathtt{q}_{\emptyset} \rangle$; and its transitions are given by:

$$\frac{\overline{q} \xrightarrow{\langle \tau, \epsilon \rangle}{}^{*}_{P} \xrightarrow{\langle \tau, l \rangle}_{P} \xrightarrow{\langle \tau, \epsilon \rangle}{}^{*}_{P} \overline{q}' \qquad q_{\mathcal{M}} \xrightarrow{\langle \tau, l \rangle}_{\mathcal{M}} q'_{\mathcal{M}}}{\langle \overline{q}, q_{\mathcal{M}} \rangle \xrightarrow{\langle \tau, l \rangle}_{P_{\mathcal{M}}} \langle \overline{q}', q'_{\mathcal{M}} \rangle}$$

In the sequel we identify concurrent systems with their induced state machines.

We can now define state robustness against a given memory subsystem. This definition essentially identifies the behaviors of a program P under a memory subsystem $\mathcal{M}$ with the first projection of the states that are reachable in $P_{\mathcal{M}}$.

Definition 2.5. A state $\overline{q}$ of a concurrent program P is *reachable under a memory subsystem* $\mathcal{M}$ if $\langle \overline{q}, q_{\mathcal{M}} \rangle$ is reachable in the concurrent system $P_{\mathcal{M}}$ for some $q_{\mathcal{M}} \in \mathcal{M}.\mathtt{Q}$.

Definition 2.6. A concurrent program P is *state robust against a memory subsystem* $\mathcal{M}$ if every reachable state of P under $\mathcal{M}$ is also reachable under SC.

We can now demonstrate the reason for including the blocking instructions `wait` and `BCAS` as primitives. Consider the following implementations of a "global barrier":

```
0 : x := 1        || 0 : y := 1
1 : r1 := y       || 1 : r2 := x          x := 1        || y := 1
2 : if r1 ≠ 1     || 2 : if r2 ≠ 1        wait(y = 1)   || wait(x = 1)
    goto 1        ||     goto 1
```

(BAR)

While the two programs are functionally equivalent, only the right program may be state robust against memory subsystems $\mathcal{M}$ that allow reading of "stale values" (such as RA and TSO). Indeed, the state in which both threads are in their last program line ($pc_1 = pc_2 = 2$) after reading 0 ($\Phi_1(r_1) = \Phi_2(r_2) = 0$) is reachable for the program on the left under such memory subsystem, but clearly not under SC. In many cases, such robustness violations are not harmful for the safety of the program, as they only imply that under weak memory the program may remain longer waiting in the busy loop.[2] A corresponding state is not reachable for the program on the right, and thus, using the blocking wait instruction, one may mask such benign robustness violations.

Similar benign robustness violations when using `CAS`, e.g., in spin loops, can be avoided using the `BCAS` primitive. Handling blocking instructions is essential to establish robustness of some interesting examples (e.g., RCU), without having more fences than actually necessary for program correctness.

3 Release/Acquire Semantics

In this section, we introduce the RA memory subsystem. RA's original presentation, as a fragment of C/C++11 [14], is declarative (a.k.a. axiomatic), i.e., it is formulated as a collection of formal consistency constraints that are used to filter candidate execution graphs. In our proofs we use such a presentation (see [1, §A])), but for the current purpose we need to define RA as an LTS. The declarative RA semantics can be easily "operationalized", as was done, e.g., in [54], so that consistent execution graphs are incrementally constructed. We will need this presentation as well (see §4.2), but since execution-graph semantics is often considered unintuitive, we present here an equivalent operational model, due to [33], which is perhaps more natural as an operational semantics for readers unfamiliar with the declarative style.

The memory in the RA operational model is a set of timestamped messages, which record all previously executed writes. *Timestamps* are taken to be natural numbers, $\mathsf{Time} \triangleq \mathbb{N}$. A timestamp and a location uniquely identify a message (that is, there cannot coexist in memory two messages of the same location and timestamp). Each thread maintains its *view* of the memory, where $T \in \mathsf{View}$ is a function $\mathsf{Loc} \to \mathsf{Time}$. The thread's view places lower bounds on the set of messages that the thread may read, as well as the timestamps it may pick when adding new messages to memory. Messages carry views as well, which record the thread's view at the time the message was added to memory. When a message is read, its view is incorporated into the thread view, which, roughly speaking, ensures that the thread becomes aware of whatever the message it reads was aware of.

Formally, a *message* $m \in \mathsf{Msg}$ is a tuple of the form $\langle x{=}v@t, T \rangle$ where $x \in \mathsf{Loc}$, $v \in \mathsf{Val}$, $t \in \mathsf{Time}$, and $T \in \mathsf{View}$. The states of the RA memory subsystem are given by $\mathsf{RA}.\mathtt{Q} \triangleq \mathcal{P}(\mathsf{Msg}) \times (\mathbb{N} \to \mathsf{View})$ (it consists of memory and thread views), with the initial state being $\mathsf{RA}.\mathtt{q}_{\emptyset} \triangleq \langle \{\langle x{=}0@0, T_0 \rangle \mid x \in \mathsf{Loc}\}, \lambda n.\, T_0 \rangle$, where $T_0 \triangleq \lambda x.\, 0$ denotes the initial view.

[2] Without liveness guarantees, this program may not terminate under weak memory semantics. In this paper, as most existing work on weak memory specification and verification, we focus on finite traces and safety properties.

$$\frac{\begin{array}{c}\neg\exists v', T'.\, \langle x{=}v'@t, T'\rangle \in M \\ \mathcal{T}(\tau)(x) < t \\ T = \mathcal{T}(\tau)[x \mapsto t] \\ M' = M \cup \{\langle x{=}v@t, T\rangle\} \\ \mathcal{T}' = \mathcal{T}[\tau \mapsto T] \\ l = \mathtt{W}(x, v)\end{array}}{\langle M, \mathcal{T}\rangle \xrightarrow{\langle \tau, l\rangle}_{\mathsf{RA}} \langle M', \mathcal{T}'\rangle} \qquad \frac{\begin{array}{c}\langle x{=}v@t, T\rangle \in M \\ \mathcal{T}(\tau)(x) \le t \\ \mathcal{T}' = \mathcal{T}[\tau \mapsto \mathcal{T}(\tau) \sqcup T] \\ l = \mathtt{R}(x, v)\end{array}}{\langle M, \mathcal{T}\rangle \xrightarrow{\langle \tau, l\rangle}_{\mathsf{RA}} \langle M, \mathcal{T}'\rangle}$$

$$\frac{\begin{array}{c}\langle x{=}v_{\mathsf{R}}@t, T_{\mathsf{R}}\rangle \in M \qquad \mathcal{T}(\tau)(x) \le t \\ \neg\exists v, T.\, \langle x{=}v@t+1, T\rangle \in M \\ T_{\mathsf{W}} = \mathcal{T}(\tau)[x \mapsto t+1] \sqcup T_{\mathsf{R}} \\ M' = M \cup \{\langle x{=}v_{\mathsf{W}}@t+1, T_{\mathsf{W}}\rangle\} \qquad \mathcal{T}' = \mathcal{T}[\tau \mapsto T_{\mathsf{W}}] \\ l = \mathtt{RMW}(x, v_{\mathsf{R}}, v_{\mathsf{W}})\end{array}}{\langle M, \mathcal{T}\rangle \xrightarrow{\langle \tau, l\rangle}_{\mathsf{RA}} \langle M', \mathcal{T}'\rangle}$$

Figure 3. Transitions of the RA memory subsystem.

The transitions of RA are given in Fig. 3, where $\sqcup$ denotes pointwise maximum ($T_1 \sqcup T_2 = \lambda x.\ \max\{T_1(x), T_2(x)\}$). To perform a write to x, thread τ (1) picks a timestamp that is available for x in the current memory and is greater than the timestamp in τ's view for x; (2) updates its view to include the new timestamp; (3) adds a message to the memory carrying τ's (updated) view. In turn, to read from x, τ may pick any message of x in the memory whose timestamp is not lower than the timestamp in τ's view for x. The view of the read message is incorporated in τ's view. Finally, RMWs are obtained as an atomic combination of a read and a write, but crucially require that the timestamp of the added message is the successor of the timestamp of the read message. This guarantees that distinct RMWs never read from the same message (see Ex. 3.5 below).

Next, we provide simple examples of runs of concurrent programs under the RA memory subsystem, and analyze their robustness. When writing views, we often write only their non-zero elements.

Example 3.1 (Store buffer)**.** The following program is the simplest example of a weak behavior allowed by RA:

$$\begin{array}{l||l} x := 1 & y := 1 \\ a := y \ /\!/\, 0 & b := x \ /\!/\, 0 \end{array} \qquad \text{(SB)}$$

Here and henceforth, we use comment annotations to denote a particular program state. In this example, the annotations denote the state in which both program counters point to the end of the program, and the values of a and b are both 0. To reach this state under RA (cf. Def. 2.5), τ_1 may run first: add $\langle x{=}1@1, [x \mapsto 1]\rangle$ to the memory (this does not affect the view of τ_2), and read the initial message $\langle y{=}0@0, T_0\rangle$. Then, τ_2 adds $\langle y{=}1@1, [y \mapsto 1]\rangle$ to the memory, and it is free read the initial message $\langle x{=}0@0, T_0\rangle$. Under SC, this state is clearly unreachable, and thus, this program is not state robust against RA (cf. Def. 2.6).

Example 3.2 (Message passing)**.** RA is designed to support "flag-based" synchronization. That is, the following annotated behavior is *disallowed* under RA:

$$\begin{array}{l||l} x := 1 & a := y \ /\!/\, 1 \\ y := 1 & b := x \ /\!/\, 0 \end{array} \qquad \text{(MP)}$$

Indeed, τ_2 can read 1 for y, only after τ_1 executed the two writes adding messages $m_x = \langle x{=}1@t_x, [x \mapsto t_x]\rangle$ and $m_y = \langle y{=}1@t_y, [x \mapsto t_x, y \mapsto t_y]\rangle$ to the memory with $t_x, t_y > 0$. When reading m_y, τ_2 increases its view of x to be t_x, and then, since $t_x > 0$, it is unable to read the initial message of x, and must read m_x. Hence, it can be easily seen that this program is state robust against RA. This example also shows that a stronger definition of robustness, which requires that P_{SC} and P_{RA} have the same traces, is too strong to be of any use. Indeed, the transition $\langle \tau_2, \mathtt{R}(y, 0)\rangle$ is allowed under RA also after τ_1 performed its two writes, and thus, such stronger condition would deem this program as non-robust.

Example 3.3 (Independent reads of independent writes)**.** Unlike TSO, RA is *non-multi-copy-atomic*. That is, different threads may observe different stores in different orders. Thus, RA allows the following behavior:

$$\begin{array}{l||l||l||l} x := 1 & a := x \ /\!/\, 1 & c := y \ /\!/\, 1 & y := 1 \\ & b := y \ /\!/\, 0 & d := x \ /\!/\, 0 & \end{array} \qquad \text{(IRIW)}$$

Indeed, nothing in RA forbids a run in which the two writers finished their execution, and then τ_2 picks the message written by τ_1 for x and the initialization message for y, while τ_3 picks the message written by τ_4 for y and the initialization message for x. The corresponding program state is unreachable under SC, and, thus, this program is not state robust against RA. (It is, nevertheless, robust against TSO.)

Example 3.4. Unlike the SRA model [36], under RA, write steps do not have to choose a *globally* maximal timestamp. Thus, the following outcome is allowed [56], and the program is not state robust against RA (it is robust against TSO):

$$\begin{array}{l||l} x := 1 & y := 1 \\ y := 2 & x := 2 \\ a := y \ /\!/\, 1 & a := x \ /\!/\, 1 \end{array} \qquad \text{(2+2W)}$$

Indeed, to execute both writes, τ_1 may add the messages $m_1^x = \langle x{=}1@2, [x \mapsto 2]\rangle$ and $m_2^y = \langle y{=}2@1, [x \mapsto 2, y \mapsto 1]\rangle$, and τ_2 may add the messages $m_1^y = \langle y{=}1@2, [y \mapsto 2]\rangle$ and $m_2^x = \langle x{=}2@1, [x \mapsto 1, y \mapsto 2]\rangle$. Now, τ_1's view for y is 1 and it may read m_1^y, and τ_2's view for x is 1 and it may read m_1^x.

Example 3.5. A crucial property of RMWs is that two (successful) RMWs never read from the same message. Indeed, this allows the standard implementation of lock acquisition using RMWs. This property is guaranteed in RA by forcing RMWs to use $t + 1$ as the timestamp for the added message, where t is the timestamp of the message that was read. To see how this works consider the following (robust) program (the annotated behavior is disallowed under RA):

$$a := \texttt{CAS}(x, 0 \to 1) \;/\!/\, 0 \;\|\; b := \texttt{CAS}(x, 0 \to 1) \;/\!/\, 0 \qquad \text{(2RMW)}$$

W.l.o.g., if τ_1 runs first, it reads from the initialization message $\langle x{=}0@0, T_0\rangle$ (it is the only message of x in the memory), and it is forced to add a message *with timestamp* 1, namely $\langle x{=}1@1, [x \mapsto 1]\rangle$. When τ_2 runs, it may *not* read from the initialization message, as it will again require adding a message of x with timestamp 1, but such a message already exists in memory. Thus, it may only read from the message that was added by τ_1, and the CAS will fail.

Example 3.6. RMW operations to a distinguished otherwise-unused location can force synchronization, practically serving as *SC-fences* [36, 37] (in fact, this is how we encode SC-fences in our programming language). To see this, consider the following modification of the SB program:

$$\begin{array}{l|l} x := 1 & y := 1 \\ r := \texttt{FADD}(f, 0) & r := \texttt{FADD}(f, 0) \\ a := y \;/\!/\, 0 & b := x \;/\!/\, 0 \end{array} \qquad \text{(SB+RMWs)}$$

Here, the annotated program behavior is disallowed under RA, and, consequently, this program is state robust against RA. Indeed, suppose, w.l.o.g., that τ_1 executes the FADD first and adds the message $m = \langle f{=}0@1, [x \mapsto t_x, f \mapsto 1]\rangle$ (where $t_x > 0$). When τ_2 executes its FADD, it has to read m, and update its view of x to t_x. Then, when it reads x it may not pick the initial message. It is crucial to use the same location in both FADDs: unlike TSO, under RA a single barrier (equivalently, a single FADD instruction to an otherwise-unused location) has no effect.

Finally, note that SC is clearly stronger than RA:

Lemma 3.7. *If a state $\overline{q}$ of a concurrent program P is reachable under* SC*, then it is also reachable under* RA.

Proof. RA can simulate SC: in read (and RMW) steps, read the message with the maximal timestamp; and in write steps, pick t to be greater than the maximal timestamp of the messages of the written location. □

4 Execution-Graph Robustness

While state robustness is a natural criterion, it is also very fragile and hard to test. For instance, if we replace the two written values in the SB program (Ex. 3.1) by 0's (writing once again the initial value), then the program becomes state robust, simply because reachable program states cannot distinguish runs under RA from runs under SC. Similarly, if we remove the two final reads in the 2+2W program (Ex. 3.4), we obtain a "vacuously" state robust program. In this section, we present a stronger notion of robustness, which we call *execution-graph robustness*. (In particular, these two examples are not execution-graph robust.) In §5, we show how execution-graph robustness can be decided. This leads to a sound verification algorithm for state robustness.

Execution-graph robustness is based on different presentations of the SC and RA memory subsystems, which we denote by SCG and RAG, whose states are execution graphs capturing (partially ordered) histories of executed actions. The fact that the states of SCG and RAG are the same mathematical objects allows us to easily compare program behaviors under the two memory subsystems. In the rest of this section, we present SCG and RAG, and define execution-graph robustness. First, we define execution graphs, starting with their nodes, called *events*.

Definition 4.1. An *event* $e \in \mathsf{Event}$ is a tuple $\langle \tau, s, l\rangle \in (\mathbb{N} \uplus \{\bot\}) \times \mathbb{N} \times \mathsf{Lab}$, where τ is a thread identifier (or $\bot$ for initialization events), s is a serial number inside each thread (0 for initialization events), and l is a label (as defined in Def. 2.1). The functions $\mathtt{tid}$, $\mathtt{sn}$, and $\mathtt{lab}$ return the thread identifier, serial number, and label 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 use R, W, RMW for the following sets of events:

$$\mathsf{R} \triangleq \{e \mid \mathtt{typ}(e) \in \{\mathtt{R}, \mathtt{RMW}\}\} \quad \mathsf{W} \triangleq \{e \mid \mathtt{typ}(e) \in \{\mathtt{W}, \mathtt{RMW}\}\}$$

$$\mathsf{RMW} \triangleq \{e \mid \mathtt{typ}(e) = \mathtt{RMW}\}$$

We employ subscripts and superscripts to restrict sets of events to certain location and thread identifier (e.g., $\mathsf{W}_x = \{w \in \mathsf{W} \mid \mathtt{loc}(w) = x\}$ and $E^\tau = \{e \in E \mid \mathtt{tid}(e) = \tau\}$).

Definition 4.2. The set Init of *initialization events* is given by $\mathsf{Init} \triangleq \{\langle \bot, 0, \mathtt{W}(x, 0)\rangle \mid x \in \mathsf{Loc}\}$. We say that a set $E \subseteq \mathsf{Event}$ is *initialized* if $\mathsf{Init} \subseteq E$, and $\mathtt{tid}(e) \neq \bot$ and $\mathtt{sn}(e) \neq 0$ for every $e \in E \setminus \mathsf{Init}$.

Our representation of events induces a *sequenced-before* partial order on events, where $e_1 < e_2$ holds iff ($e_1 \in \mathsf{Init}$ and $e_2 \notin \mathsf{Init}$) or ($e_1 \notin \mathsf{Init}$, $e_2 \notin \mathsf{Init}$, $\mathtt{tid}(e_1) = \mathtt{tid}(e_2)$, and $\mathtt{sn}(e_1) < \mathtt{sn}(e_2)$). That is, initialization events precede all non-initialization events, while events of the same thread are ordered according to their serial numbers.

In turn, an execution graph consists of a set of events, a *reads-from* mapping that determines the write event from which each read reads its value, and a *modification order* which totally orders the writes to each location. In terms of the model in §3, the modification order represents the timestamp order on messages to each location.

Definition 4.3. An *execution* graph $G \in \mathsf{EGraph}$ is a tuple $\langle E, \mathit{rf}, \mathit{mo}\rangle$ where:

1. E is an initialized finite set of events.
2. rf, called *reads-from*, is a relation on E satisfying:
   - If $\langle w, r\rangle \in \mathit{rf}$ then $w \in \mathsf{W}$, $r \in \mathsf{R}$, $\mathtt{loc}(w) = \mathtt{loc}(r)$, $\mathtt{val}_\mathtt{W}(w) = \mathtt{val}_\mathtt{R}(r)$, and $w \neq r$.
   - $w_1 = w_2$ whenever $\langle w_1, r\rangle, \langle w_2, r\rangle \in \mathit{rf}$ (each read reads from at most one write).
   - $E \cap \mathsf{R} \subseteq \mathit{codom}(\mathit{rf})$ (each read reads from some write).
3. mo, called *modification order*, is a disjoint union of relations $\{\mathit{mo}_x\}_{x \in \mathsf{Loc}}$, such that each mo_x is a strict total order on $E \cap \mathsf{W}_x$.

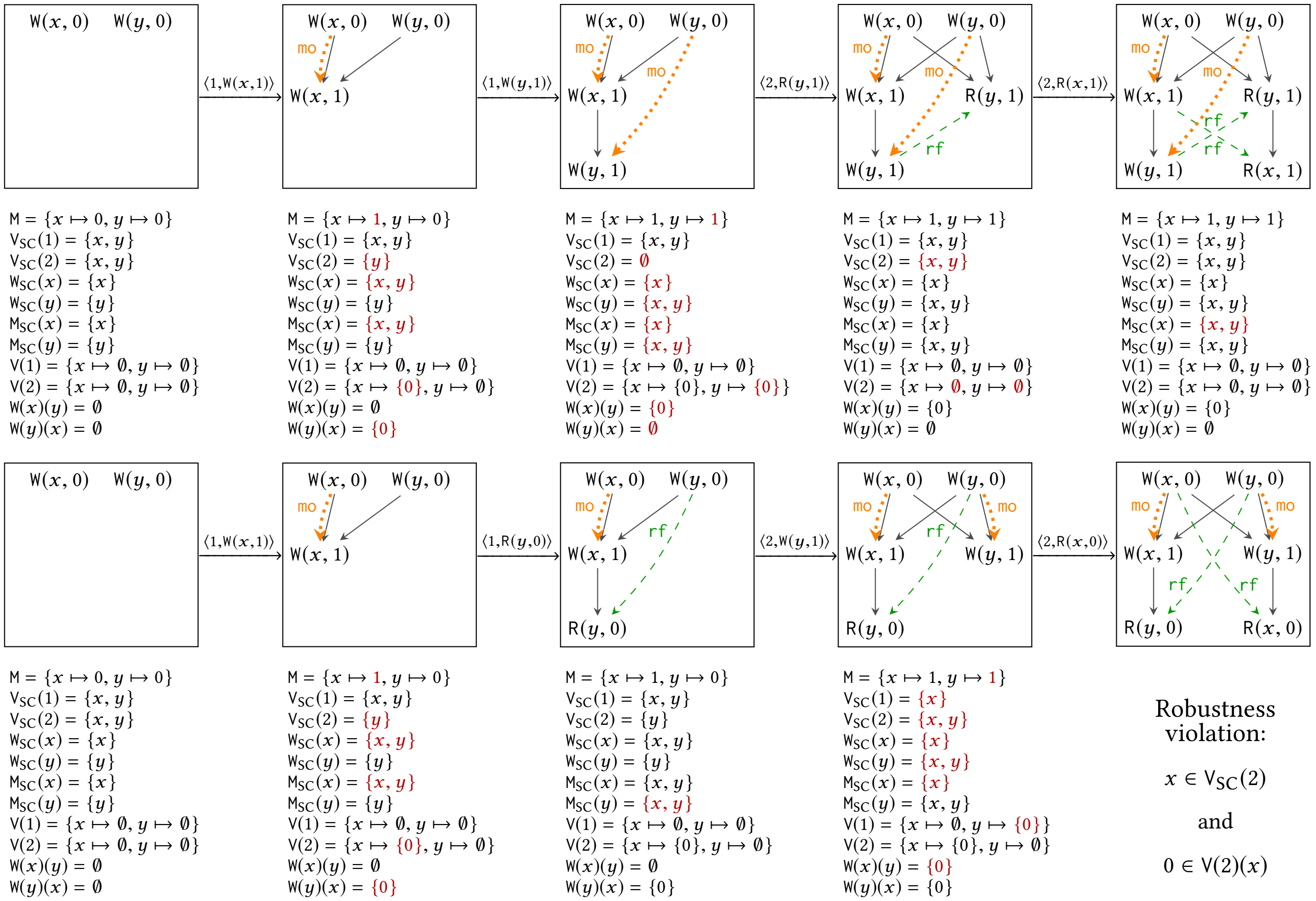

Figure 4. Illustrations of runs: (*i*) of SCG for the MP program (Ex. 3.2); and (*ii*) of RAG for the SB program (Ex. 3.1). Each illustration is followed by the corresponding run of SCM for monitoring robustness as described in §5 (deltas from the previous state are colored).

We denote the components of G by G.E, G.rf and G.mo. We use G.po to denote the restriction of the order on events to G.E ($G.\mathsf{po} \triangleq [G.\mathsf{E}];<;[G.\mathsf{E}]$). In addition, for a set $E' \subseteq$ Event, we write $G.E'$ for $G.\mathsf{E} \cap E'$ (e.g., $G.\mathsf{W} = G.\mathsf{E} \cap \mathsf{W}$).

Next, we define a general execution-graph-based memory subsystem, called FG (standing for "Free Graphs"). Later, the memory subsystems SCG and RAG are defined as restrictions of FG. To define FG, we use the following notation that extends a given graph G with a new event e, placed last in its thread, and either reading from a designated event w or placed as the immediate successor of w in the modification order. When e is an RMW event, it is *both* reading from w, and placed as the immediate successor of w in the modification order. This is in accordance with the usual atomicity restriction in declarative semantics, according to which RMWs read from their immediate mo-predecessors.

Notation 4.4. *For* $G \in$ EGraph, $\tau \in \mathbb{N}$, $l \in$ Lab *and* $w \in$ W, $\mathsf{add}(G, \tau, l, w)$ *denotes the triple* $\langle E', rf', mo' \rangle$ *defined as follows, where* $e = \langle \tau, \max\{\mathsf{sn}(e) \mid e \in G.\mathsf{E}^\tau\} + 1, l\rangle$*:*

$$E' = G.\mathsf{E} \cup \{e\} \qquad rf' = \begin{cases} G.\mathsf{rf} \cup \{\langle w, e\rangle\} & e \in \mathsf{R} \\ G.\mathsf{rf} & \textit{otherwise} \end{cases}$$

$$mo' = \begin{cases} G.\mathsf{mo} \cup \mathit{dom}(G.\mathsf{mo}^? \,;[\{w\}]) \times \{e\} & \\ \quad \cup \{e\} \times \mathit{codom}([\{w\}] \,; G.\mathsf{mo}) & e \in \mathsf{W} \\ G.\mathsf{mo} & \textit{otherwise} \end{cases}$$

Definition 4.5. The initial execution graph $G_\emptyset$ is given by $G_\emptyset \triangleq \langle \mathsf{Init}, \emptyset, \emptyset\rangle$. The memory subsystem FG is defined by $\mathsf{FG.Q} \triangleq$ EGraph, $\mathsf{FG.q}_\emptyset \triangleq G_\emptyset$, and $\rightarrow_{\mathsf{FG}}$ is defined as follows:

$$\frac{w \in G.\mathsf{W}_{\mathsf{loc}(l)} \qquad \mathsf{typ}(l) \in \{\mathsf{R}, \mathsf{RMW}\} \implies \mathsf{val_W}(w) = \mathsf{val_R}(l)}{G \xrightarrow{\langle \tau, l\rangle}_{\mathsf{FG}} \mathsf{add}(G, \tau, l, w)}$$

The conditions in FG's step ensure that $\mathsf{add}(G, \tau, l, w)$ is indeed an execution graph: mo should only relate events in W of the same location; and rf goes from W to R only between events of the same location and matching values. Below, we refer to the write w in such steps as the *predecessor write*.

Next, we present the memory subsystems SCG and RAG. Both are *based on execution graphs*: $\mathsf{SCG.Q} = \mathsf{RAG.Q} \triangleq \mathsf{EGraph}$; $\mathsf{SCG.q_0} = \mathsf{RAG.q_0} \triangleq G_0$; and $\xrightarrow{\langle\tau,l\rangle}_{\mathsf{SCG}} \subseteq \xrightarrow{\langle\tau,l\rangle}_{\mathsf{FG}}$ and $\xrightarrow{\langle\tau,l\rangle}_{\mathsf{RAG}} \subseteq \xrightarrow{\langle\tau,l\rangle}_{\mathsf{FG}}$ for every $\tau \in \mathbb{N}$ and $l \in \mathsf{Lab}$.

4.1 The Memory Subsystem SCG

The steps of SCG are uniformly given by:

$$\frac{\mathtt{typ}(l) \in \{\mathtt{R}, \mathtt{RMW}\} \implies \mathtt{val_W}(G.w^{\max}_{\mathtt{loc}(l)}) = \mathtt{val_R}(l)}{G \xrightarrow{\langle\tau,l\rangle}_{\mathsf{SCG}} \mathtt{add}(G, \tau, l, G.w^{\max}_{\mathtt{loc}(l)})}$$

where $G.w^{\max}_x$ denotes the $G.\mathtt{mo}$ maximal write to x in G ($G.w^{\max}_x \triangleq \max_{G.\mathtt{mo}} G.\mathsf{W}_x$).

SCG steps require the predecessor write to be $G.w^{\max}_{\mathtt{loc}(l)}$: added write events are placed last in $G.\mathtt{mo}$, and read events read from the latest added write. Figure 4 illustrates an example of a run of the MP program (Ex. 3.2) under SCG.

Lemma 4.6. SCG *and* SC *have the same traces.*

Proof (outline). Define the *memory* of a given $G \in \mathsf{EGraph}$ by $M(G) \triangleq \lambda x.\, \mathtt{val_W}(G.w^{\max}_x)$. It is easy to show that $\mathsf{SC.q_0} = M(G_0)$ and $\{\langle M(G), G\rangle \mid G \in \mathsf{EGraph}\}$ is a bisimulation relation between SC and SCG. □

4.2 The Memory Subsystem RAG

To define the transitions of RAG, we use the following standard derived "happens-before" relation:

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

Roughly speaking, $G.\mathtt{hb}$ abstracts RA's execution order: any run of the timestamp machine in §3 follows some linearization of hb, and, conversely, all linearizations of hb induce runs of the timestamp machine. Using hb, the steps of RAG are uniformly given by:

$$\frac{\begin{array}{c} w \in G.\mathsf{W}_{\mathtt{loc}(l)} \\ \mathtt{typ}(l) \in \{\mathtt{R}, \mathtt{RMW}\} \implies \mathtt{val_W}(w) = \mathtt{val_R}(l) \\ w \notin dom(G.\mathtt{mo} \,;\, G.\mathtt{hb}^? \,;\, [G.\mathsf{E}^\tau]) \\ \mathtt{typ}(l) \in \{\mathtt{W}, \mathtt{RMW}\} \implies w \notin dom(G.\mathtt{mo}|_{\mathrm{imm}} \,;\, [\mathsf{RMW}]) \end{array}}{G \xrightarrow{\langle\tau,l\rangle}_{\mathsf{RAG}} \mathtt{add}(G, \tau, l, w)}$$

The first two conditions in the step are the general conditions of FG (see Def. 4.5). The third and fourth conditions restrict the choice of the predecessor write w. Unlike in SCG, w is not necessarily $G.w^{\max}_{\mathtt{loc}(l)}$. Instead, it is subject to two conditions. First, the thread that takes the action must not have observed an mo-later write, where observed writes are writes that have a (possibly empty) hb-path to (some event of) the thread ($w \notin dom(G.\mathtt{mo} \,;\, G.\mathtt{hb}^? \,;\, [G.\mathsf{E}^\tau])$). Referring to the timestamp machine, this is in accordance with the choice of the message to read in read steps and the new added messages in write steps (their timestamp cannot be smaller than the last timestamp observed by the thread for the location). Second, when writing (by a write or an RMW), the predecessor write w cannot be the immediate mo-predecessor of some (other) RMW event ($w \notin dom(G.\mathtt{mo}|_{\mathrm{imm}} \,;\, [\mathsf{RMW}])$). In the timestamp machine, this corresponds to the fact that timestamp of the message added by an RMW must be the immediate successor of the timestamp of the message read by the RMW. Note that in graphs generated by RAG, RMWs always read from their immediate mo-predecessor ($G.\mathtt{rf} ; [\mathsf{RMW}] = G.\mathtt{mo}|_{\mathrm{imm}} ; [\mathsf{RMW}]$), which is the usual atomicity condition in declarative weak memory semantics.

It is easy to see that SCG is more restrictive than RAG (and thus, the run of SCG for the MP program in Fig. 4 is also allowed under RAG):

Lemma 4.7. *If* $G \xrightarrow{\langle\tau,l\rangle}_{\mathsf{SCG}} G'$, *then* $G \xrightarrow{\langle\tau,l\rangle}_{\mathsf{RAG}} G'$.

Proof. Pick $w = G.w^{\max}_{\mathtt{loc}(l)}$ as the predecessor write. By definition we have $w \in G.\mathsf{W}_{\mathtt{loc}(l)}$, $w \notin dom(G.\mathtt{mo} \,;\, G.\mathtt{hb}^? \,;\, [G.\mathsf{E}^\tau])$, and $w \notin dom(G.\mathtt{mo}|_{\mathrm{imm}} \,;\, [\mathsf{RMW}])$. □

Figure 4 illustrates an example of a run of the SB program (Ex. 3.1) under RAG. The last step there is disallowed by SCG—the predecessor write is not the mo-maximal one.

Next, we state the equivalence between RAG and RA. A proof outline is provided in [1, §B].

Lemma 4.8. RAG *and* RA *have the same traces.*

4.3 Execution-Graph Robustness

Next, we define execution-graph robustness and show that it implies state robustness.

Definition 4.9. A concurrent program P is *execution-graph robust* against RA if every reachable state $\langle \overline{q}, G\rangle$ in the concurrent system P_{RAG} is also reachable in P_{SCG}.

Proposition 4.10. *If* P *is execution-graph robust against* RA *then it is state robust against* RA.

Proof. Let $\overline{q}$ be a state of P that is reachable under RA. Let $\langle M, \mathcal{T}\rangle \in \mathsf{RA.Q}$ such that $\langle \overline{q}, \langle M, \mathcal{T}\rangle\rangle$ is reachable in P_{RA}. By Lemma 4.8, $\langle \overline{q}, G\rangle$ is reachable in P_{RAG} for some G. Since P is execution-graph robust against RA, it follows that $\langle \overline{q}, G\rangle$ is reachable in P_{SCG}. By Lemma 4.6, $\langle \overline{q}, M\rangle$ is reachable in P_{SC} for some $M \in \mathsf{SC.Q}$, and so $\overline{q}$ is reachable under SC. □

Execution-graph robustness, as we demonstrate below, is not overly strong for establishing state robustness in a variety of concurrent algorithms. In particular, the state robust litmus tests mentioned in §3 (MP,2RMW,SB+RMWs) are also execution-graph robust.

5 Verifying Execution-Graph Robustness

In this section, we present our approach to the verification of execution-graph robustness against RA. First, Thm. 5.1 below reduces this problem to reachability of certain configurations in P_{SCG}. To state this theorem, we require two more standard derived relations in execution graphs:

$$G.\mathtt{fr} \triangleq (G.\mathtt{rf}^{-1} \,;\, G.\mathtt{mo}) \setminus [G.\mathsf{E}] \quad (\textit{from-read/reads-before})$$

$$G.\mathtt{hb}_{\mathsf{SC}} \triangleq (G.\mathtt{hb} \cup G.\mathtt{mo} \cup G.\mathtt{fr})^{+} \qquad (\text{SC-}\textit{happens-before})$$

The *from-read* relation, $\mathtt{fr}$, relates every read event r to all writes that are $\mathtt{mo}$-later than the write that r reads from (identity is subtracted to avoid self loops in RMW events). The SC-*happens-before* relation, $G.\mathtt{hb}_{\mathsf{SC}}$, following [51], abstracts SC's execution order: to yield certain execution G, the SCG memory subsystem must follow $G.\mathtt{hb}_{\mathsf{SC}}$. Thus, runs of SCG can yield an execution graph G iff $G.\mathtt{hb}_{\mathsf{SC}}$ is irreflexive.

Theorem 5.1. *Let P be a concurrent program. Call a tuple $\langle \overline{q}, G, \tau, l, w\rangle \in P.\mathsf{Q} \times \mathsf{EGraph} \times \mathsf{Tid} \times \mathsf{Lab} \times \mathsf{Event}$ a non-robustness witness for P if the following hold:*

- *$\langle \overline{q}, G\rangle$ is reachable in the concurrent system P_{SCG}.*
- *$\overline{q}$ enables $\langle \tau, l\rangle$ (in the LTS induced by P).*
- *$w \neq G.w^{\max}_{\mathtt{loc}(l)}$.*
- *$G \xrightarrow{\langle \tau, l\rangle}_{\mathsf{RAG}} \mathsf{add}(G, \tau, l, w)$.*
- *$G.w^{\max}_{\mathtt{loc}(l)} \in dom(G.\mathtt{hb}_{\mathsf{SC}} \,;\, [G.\mathsf{E}^{\tau}])$.*

Then, P is execution-graph robust against RA *iff there does not exist a non-robustness witness for P.*

Theorem 5.1 reduces execution-graph robustness of a program P to the existence of a reachable state in the concurrent system P_{SCG} that satisfies certain properties. More precisely, P is not robust iff there exist a reachable state $\langle \overline{q}, G\rangle$ of P_{SCG} and a transition $\langle \tau, l\rangle$ that is enabled in $\overline{q}$, such that: (a) there is an $\mathtt{hb}_{\mathsf{SC}}$-path in G from $w^{\max}_{\mathtt{loc}(l)}$ to (some event of) thread τ; and (b) G enables the transition $\langle \tau, l\rangle$ in RAG with a predecessor write $w \neq G.w^{\max}_{\mathtt{loc}(l)}$.

The proof is given in [1, §A]. Roughly speaking, we utilize purely declarative presentations of SCG and RAG, and show that the existence of a non-robustness witness allows RA executions to diverge w.r.t. SC ones, and that given a "minimal" such divergence, one can construct a non-robustness witness. The latter has generally a similar structure to proofs establishing the DRF (data-race-freedom) guarantee [6, 29].

We note that DRF for RA can be easily obtained as a corollary of Thm. 5.1. Indeed, if a program P is race-free (under SC), then all reachable states $\langle \overline{q}, G\rangle$ in P_{SCG} satisfy $G.\mathtt{mo} \cup G.\mathtt{fr} \subseteq G.\mathtt{hb}$. It follows that $G.\mathtt{hb}_{\mathsf{SC}} \subseteq G.\mathtt{hb}$, and thus, $G.w^{\max}_{\mathtt{loc}(l)} \in dom(G.\mathtt{hb}_{\mathsf{SC}} \,;\, [G.\mathsf{E}^{\tau}])$ implies that only $G.w^{\max}_{\mathtt{loc}(l)}$ may serve as the predecessor write in RAG transitions from G. Therefore, P cannot have a non-robustness witness, and Thm. 5.1 ensures that it is execution-graph robust.

Similarly, it follows that a program with no concurrent writes under SC cannot have weak behaviors allowed by RA (as was established in [7] for a certain variant of causal consistency). Indeed, if P has no concurrent writes (under SC), then all reachable states $\langle \overline{q}, G\rangle$ in P_{SCG} satisfy $[\mathsf{W}] \,;\, G.\mathtt{hb}_{\mathsf{SC}} \subseteq G.\mathtt{hb}$ (use $\mathtt{hb}$ to reach the last write in the $\mathtt{hb}_{\mathsf{SC}}$-path and from that point on no $\mathtt{mo}$ and $\mathtt{fr}$ edges are used). Again, Thm. 5.1 ensures that P is execution-graph robust.

It remains to show that the condition in Thm. 5.1 can be automatically checked. Since SCG is not finite (execution graphs of programs with loops may grow unboundedly), we cannot naively explore traces of P_{SCG}. The key idea is to define a *finite* memory subsystem, which we call SCM (for SC with Monitors), that simulates SCG (so that they have the same traces) and precisely track the properties of SCG's execution graphs that are needed for monitoring the above condition.

Next, we gradually present SCM's states, which are composed of eight components in total, and the transitions between them. Figure 4 provides detailed examples of runs of SCM for the MP and SB programs, together with the corresponding runs of SCG. Below, we use I as a metavariable for states of SCM and write $I(G)$ for the SCM state that corresponds to an execution graph G.

***Memory* ($I.\mathsf{M}$).** The basic building block for SCM is the (finite) memory subsystem SC whose states are simple location-value mappings (see §2.3). Thus, a state I of SCM has a memory component, denoted $I.\mathsf{M}$, which is a function from Loc to Val storing the value written by $G.w^{\max}_{x}$ for every location x. Formally, we have

$$I(G).\mathsf{M} = \lambda x.\, \mathtt{val}_{\mathtt{W}}(G.w^{\max}_{x}).$$

The transitions of SCM are subject to the same constraints as SC with respect to this component. The other components of the states of SCM are used to track more properties of G, and do not restrict SCM's traces. Thus, the fact that SCM has the same traces as SCG directly follows from Lemma 4.6.

$\mathtt{hb}_{\mathsf{SC}}$*-tracking* ($I.\mathsf{V}_{\mathsf{SC}}, I.\mathsf{M}_{\mathsf{SC}}, I.\mathsf{W}_{\mathsf{SC}}$). For checking condition (a) above, we need to know for every thread τ and location x whether τ is "$\mathtt{hb}_{\mathsf{SC}}$-aware" of $w^{\max}_{x}$. To include and maintain this information in a state I of SCM, we use three components, denoted by $I.\mathsf{V}_{\mathsf{SC}}$, $I.\mathsf{M}_{\mathsf{SC}}$ and $I.\mathsf{W}_{\mathsf{SC}}$.

The first, $I.\mathsf{V}_{\mathsf{SC}}$, is a function in $\mathsf{Tid} \to \mathcal{P}(\mathsf{Loc})$ tracking precisely this property. Formally, we have:

$$I(G).\mathsf{V}_{\mathsf{SC}} = \lambda \tau.\, \{x \mid G.w^{\max}_{x} \in dom(G.\mathtt{hb}_{\mathsf{SC}}{}^{?} \,;\, [\mathsf{Init} \cup G.\mathsf{E}^{\tau}])\}.$$

Having $x \in I(G).\mathsf{V}_{\mathsf{SC}}(\tau)$ means that τ is $\mathtt{hb}_{\mathsf{SC}}$-aware of $w^{\max}_{x}$, i.e., $G.w^{\max}_{x}$ is an initialization write (of which all threads are aware) or $\langle G.w^{\max}_{x}, e\rangle \in G.\mathtt{hb}_{\mathsf{SC}}{}^{?}$ for some $e \in G.\mathsf{E}^{\tau}$.

In turn, to maintain $I.\mathsf{V}_{\mathsf{SC}}$, we include two additional components, $I.\mathsf{M}_{\mathsf{SC}}$ and $I.\mathsf{W}_{\mathsf{SC}}$, both of which are functions in $\mathsf{Loc} \to \mathcal{P}(\mathsf{Loc})$. Consider first an SCG-step that adds a write (or RMW) event w to location x in thread τ. Following SCG, w is placed last in $\mathtt{mo}$, which means that every event accessing x becomes $\mathtt{hb}_{\mathsf{SC}}$ before w (writes to x have $\mathtt{mo}$ to w and reads from x have $\mathtt{fr}$ to w). In turn, the thread τ in which w is added will have (additional) $\mathtt{hb}_{\mathsf{SC}}$-paths from every $w^{\max}_{y}$ that previously had an $\mathtt{hb}_{\mathsf{SC}}$-path to some event accessing x.

	$\langle \tau, \mathsf{W}(x, v)\rangle$ or $\langle \tau, \mathsf{RMW}(x, v_{\mathsf{R}}, v_{\mathsf{W}})\rangle$	$\langle \tau, \mathsf{R}(x, v)\rangle$
$\mathsf{V}'_{\mathsf{SC}} = \lambda \pi.$	$\begin{cases} \mathsf{V}_{\mathsf{SC}}(\tau) \cup \mathsf{M}_{\mathsf{SC}}(x) & \pi = \tau \\ \mathsf{V}_{\mathsf{SC}}(\pi) \setminus \{x\} & \pi \neq \tau \end{cases}$	$\begin{cases} \mathsf{V}_{\mathsf{SC}}(\tau) \cup \mathsf{W}_{\mathsf{SC}}(x) & \pi = \tau \\ \mathsf{V}_{\mathsf{SC}}(\pi) & \pi \neq \tau \end{cases}$
$\mathsf{M}'_{\mathsf{SC}} = \lambda y.$	$\begin{cases} \mathsf{M}_{\mathsf{SC}}(x) \cup \mathsf{V}_{\mathsf{SC}}(\tau) & y = x \\ \mathsf{M}_{\mathsf{SC}}(y) \setminus \{x\} & y \neq x \end{cases}$	$\begin{cases} \mathsf{M}_{\mathsf{SC}}(x) \cup \mathsf{V}_{\mathsf{SC}}(\tau) & y = x \\ \mathsf{M}_{\mathsf{SC}}(y) & y \neq x \end{cases}$
$\mathsf{W}'_{\mathsf{SC}} = \lambda y.$	$\begin{cases} \mathsf{M}_{\mathsf{SC}}(x) \cup \mathsf{V}_{\mathsf{SC}}(\tau) & y = x \\ \mathsf{W}_{\mathsf{SC}}(y) \setminus \{x\} & y \neq x \end{cases}$	$\mathsf{W}_{\mathsf{SC}}(y)$

Figure 5. Maintaining V_{SC}, M_{SC} and W_{SC} in SCM transitions.

To properly reflect this in $I.\mathsf{V}_{\mathsf{SC}}(\tau)$, we maintain $I.\mathsf{M}_{\mathsf{SC}}$ that tracks for every $x \in \mathsf{Loc}$ the set of locations y such that $w_y^{\max}$ has an $\mathsf{hb}_{\mathsf{SC}}$-path to some event accessing x. In steps that write to x in thread τ, we incorporate $I.\mathsf{M}_{\mathsf{SC}}(x)$ into $I.\mathsf{V}_{\mathsf{SC}}(\tau)$.

Second, similarly, when an SCG-step adds a read event r of location x in thread τ, it reads from $w_x^{\max}$, and so we have $\langle w_x^{\max}, r\rangle \in \mathsf{hb}_{\mathsf{SC}}$. In turn, thread τ will have (additional) $\mathsf{hb}_{\mathsf{SC}}$-paths from every $w_y^{\max}$ that previously had an $\mathsf{hb}_{\mathsf{SC}}$-path to $w_x^{\max}$. Accordingly, the $I.\mathsf{W}_{\mathsf{SC}}$ component tracks for every $x \in \mathsf{Loc}$ the set of locations y such that $G.w_y^{\max}$ has an $\mathsf{hb}_{\mathsf{SC}}$-path to $w_x^{\max}$. In steps that read from x in thread τ, we incorporate $I.\mathsf{W}_{\mathsf{SC}}(x)$ into $I.\mathsf{V}_{\mathsf{SC}}(\tau)$. Note that, while $y \in I.\mathsf{M}_{\mathsf{SC}}(x)$ iff $w_y^{\max}$ has an $\mathsf{hb}_{\mathsf{SC}}$-path to *some event* accessing x, we have $y \in I.\mathsf{W}_{\mathsf{SC}}(x)$ iff $w_y^{\max}$ has an $\mathsf{hb}_{\mathsf{SC}}$-path to $w_x^{\max}$ (equivalently, to some *write* event accessing x). This implies, in particular, that we always have $I.\mathsf{W}_{\mathsf{SC}}(x) \subseteq I.\mathsf{M}_{\mathsf{SC}}(x)$.

Formally, the meaning of these two "helper" components is given by:

$$I(G).\mathsf{M}_{\mathsf{SC}} = \lambda x.\, \{y \mid G.w_y^{\max} \in \mathit{dom}(G.\mathsf{hb}_{\mathsf{SC}}^{?}\,;[G.\mathsf{E}_x])\}$$
$$I(G).\mathsf{W}_{\mathsf{SC}} = \lambda x.\, \{y \mid \langle G.w_y^{\max}, G.w_x^{\max}\rangle \in G.\mathsf{hb}_{\mathsf{SC}}^{?}\}$$

Initially, we take $\mathsf{SCM}.\mathsf{q}_0.\mathsf{V}_{\mathsf{SC}} = \lambda\tau.\,\mathsf{Loc}$ and $\mathsf{SCM}.\mathsf{q}_0.\mathsf{M}_{\mathsf{SC}} = \mathsf{SCM}.\mathsf{q}_0.\mathsf{W}_{\mathsf{SC}} = \lambda x.\, \{x\}$.

Figure 5 presents the maintenance of $I.\mathsf{V}_{\mathsf{SC}}$, $I.\mathsf{M}_{\mathsf{SC}}$ and $I.\mathsf{W}_{\mathsf{SC}}$ (primed components denote the corresponding components after the transition mentioned in the column headers). In particular, note that when a write (or RMW) to x is performed it becomes the new $w_x^{\max}$ and it has no $\mathsf{hb}_{\mathsf{SC}}$-paths to other events in the execution graph. Thus, we remove x from $I.\mathsf{V}_{\mathsf{SC}}(\pi)$ for every thread π except for the one that performed the write, as well as from $I.\mathsf{M}_{\mathsf{SC}}(y)$ and $I.\mathsf{W}_{\mathsf{SC}}(y)$ for every $y \neq x$. In addition, when accessing location x in thread τ, $I.\mathsf{M}_{\mathsf{SC}}(x)$ inherits $I.\mathsf{V}_{\mathsf{SC}}(\tau)$ (every event that had $\mathsf{hb}_{\mathsf{SC}}$-path to thread τ now has $\mathsf{hb}_{\mathsf{SC}}$-path to an event accessing x), and when writing to location x in thread τ, $I.\mathsf{W}_{\mathsf{SC}}(x)$ inherits $I.\mathsf{V}_{\mathsf{SC}}(\tau)$ (every event that had $\mathsf{hb}_{\mathsf{SC}}$-path to thread τ now has $\mathsf{hb}_{\mathsf{SC}}$-path to $w_x^{\max}$).

RAG-***tracking*** ***($I.\mathsf{V}$, $I.\mathsf{W}$, $I.\mathsf{V}_{\mathsf{RMW}}$, $I.\mathsf{W}_{\mathsf{RMW}}$).*** It remains to extend the instrumentation, so that we can check for every thread τ and label l, whether the transition $\langle \tau, l\rangle$ in enabled in RAG with a predecessor write that is not $w_{\mathsf{loc}(l)}^{\max}$ (condition *(b)* above). For this matter, we include four additional components in the state I of SCM. Two of them, $I.\mathsf{V}$ and $I.\mathsf{V}_{\mathsf{RMW}}$, are functions in $(\mathsf{Tid} \times \mathsf{Loc}) \to \mathcal{P}(\mathsf{Val})$ and are the ones used to check the above condition. The other two, $I.\mathsf{W}$ and $I.\mathsf{W}_{\mathsf{RMW}}$, are functions in $(\mathsf{Loc} \times \mathsf{Loc}) \to \mathcal{P}(\mathsf{Val})$ and, as before, are used to properly maintain $I.\mathsf{V}$ and $I.\mathsf{V}_{\mathsf{RMW}}$.

To understand these components, recall the transition of RAG in §4.2:

Read: Consider first a read transition $\langle \tau, l\rangle$ with $l = \mathsf{R}(x, v_{\mathsf{R}})$. By definition, an execution graph G enables $\langle \tau, l\rangle$ with a predecessor write $w \in G.\mathsf{W}_x$ if $\mathtt{val}_{\mathsf{W}}(w) = v_{\mathsf{R}}$ and $w \notin \mathit{dom}(G.\mathsf{mo}\,; G.\mathsf{hb}^{?}\,; [G.\mathsf{E}^{\tau}])$. To be able to check this condition, we use $I.\mathsf{V}$ to track for every $\tau \in \mathsf{Tid}$ and $x \in \mathsf{Loc}$ the set of values that are written by some $w \in G.\mathsf{W}_x$ that is not $G.w_x^{\max}$ and satisfies $w \notin \mathit{dom}(G.\mathsf{mo}\,; G.\mathsf{hb}^{?}\,; [G.\mathsf{E}^{\tau}])$. Then, to check condition *(b)* above, we check whether $v_{\mathsf{R}} \in I.\mathsf{V}(\tau)(x)$. In other words, $I.\mathsf{V}(\tau)(x)$ tracks the set of values that can be read by thread τ from x under RAG, excluding the case of reading from $w_x^{\max}$ (which is also allowed by SCG).

As before, to maintain $I.\mathsf{V}$, we use another component in I. When thread τ reads (or performs an RMW to) x in a transition of SCG, it induces an $\mathsf{mo}\,;\mathsf{hb}$-path to thread τ from any write that had $\mathsf{mo}\,;\mathsf{hb}$-path to $w_x^{\max}$. Thus, after such transition, $I.\mathsf{V}(\tau)(y)$ should be restricted to values written by some $w \in G.\mathsf{W}_y$ such that $\langle w, G.w_x^{\max}\rangle \notin G.\mathsf{mo}\,; G.\mathsf{hb}^{?}$. Accordingly, $I.\mathsf{W}$ tracks for every pair $x, y \in \mathsf{Loc}$ the set of values that are written by some write $w \in G.\mathsf{W}_y$ that is not $G.w_y^{\max}$ and satisfies $\langle w, G.w_x^{\max}\rangle \notin G.\mathsf{mo}\,; G.\mathsf{hb}^{?}$.

Write and RMW: A write (or RMW) transition is similar, but it is subject to an additional constraint in RAG: the predecessor write w should not be an mo-immediate predecessor of an RMW event in G (equivalently, w should not be read by an RMW event). For this condition, we use $I.\mathsf{V}_{\mathsf{RMW}}$, that, as $I.\mathsf{V}$, tracks for every $\tau \in \mathsf{Tid}$ and $x \in \mathsf{Loc}$ the set of values that are written by some $w \in G.\mathsf{W}_x$ that is not $G.w_x^{\max}$ and satisfies $w \notin \mathit{dom}(G.\mathsf{mo}\,; G.\mathsf{hb}^{?}\,; [G.\mathsf{E}^{\tau}])$, but further requires that $w \notin \mathit{dom}(G.\mathsf{mo}|_{\mathrm{imm}}\,; [\mathsf{RMW}])$. To maintain $I.\mathsf{V}_{\mathsf{RMW}}$, we use $I.\mathsf{W}_{\mathsf{RMW}}$, which is similar to $I.\mathsf{W}$ with the same additional condition on w (i.e., $w \notin \mathit{dom}(G.\mathsf{mo}|_{\mathrm{imm}}\,; [\mathsf{RMW}])$).

Formally, the meaning of these components is given by:

$$I(G).\mathsf{V} = \lambda\tau, x.\, \mathtt{val}_{\mathsf{W}}[W \setminus \mathit{dom}(R\,; [G.\mathsf{E}^{\tau}])]$$
$$I(G).\mathsf{W} = \lambda y, x.\, \mathtt{val}_{\mathsf{W}}[W \setminus \mathit{dom}(R\,; [\{G.w_y^{\max}\}])]$$
$$I(G).\mathsf{V}_{\mathsf{RMW}} = \lambda\tau, x.\, \mathtt{val}_{\mathsf{W}}[W \setminus \mathit{dom}(R\,; [G.\mathsf{E}^{\tau}] \cup R_{\mathsf{RMW}})]$$
$$I(G).\mathsf{W}_{\mathsf{RMW}} = \lambda y, x.\, \mathtt{val}_{\mathsf{W}}[W \setminus \mathit{dom}(R\,; [\{G.w_y^{\max}\}] \cup R_{\mathsf{RMW}})]$$

where $W = G.\mathsf{W}_x \setminus \{G.w_x^{\max}\}$, $R = G.\mathsf{mo}\,; G.\mathsf{hb}^{?}$ and $R_{\mathsf{RMW}} = G.\mathsf{mo}|_{\mathrm{imm}}\,; [\mathsf{RMW}]$ (the function $\mathtt{val}_{\mathsf{W}}$ is extended to sets of events in the obvious way). Initially, since each location has only one write in the initial graph, these four components all return the empty set of values. Figure 6 presents our maintenance of these components.

	$\langle \tau, \mathsf{W}(x, v) \rangle$ where $v_\mathsf{R} = \mathsf{M}(x)$	$\langle \tau, \mathsf{R}(x, v) \rangle$	$\langle \tau, \mathsf{RMW}(x, v_\mathsf{R}, v_\mathsf{W}) \rangle$
$\mathsf{V}' = \lambda \pi, y.$	$\begin{cases} \emptyset & \pi = \tau,\ y = x \\ \mathsf{V}(\pi)(x) \cup \{v_\mathsf{R}\} & \pi \neq \tau,\ y = x \\ \mathsf{V}(\pi)(y) & y \neq x \end{cases}$	$\begin{cases} \mathsf{V}(\tau)(y) \cap \mathsf{W}(x)(y) & \pi = \tau \\ \mathsf{V}(\pi)(y) & \pi \neq \tau \end{cases}$	$\begin{cases} \mathsf{V}(\tau)(y) \cap \mathsf{W}(x)(y) & \pi = \tau \\ \mathsf{V}(\pi)(x) \cup \{v_\mathsf{R}\} & \pi \neq \tau,\ y = x \\ \mathsf{V}(\pi)(y) & \pi \neq \tau,\ y \neq x \end{cases}$
$\mathsf{W}' = \lambda z, y.$	$\begin{cases} \mathsf{V}(\tau)(y) & z = x,\ y \neq x \\ \mathsf{W}(z)(x) \cup \{v_\mathsf{R}\} & z \neq x,\ y = x \\ \mathsf{W}(z)(y) & \text{otherwise} \end{cases}$	$\mathsf{W}(z)(y)$	$\begin{cases} \mathsf{W}(x)(y) \cap \mathsf{V}(\tau)(y) & z = x,\ y \neq x \\ \mathsf{W}(z)(x) \cup \{v_\mathsf{R}\} & z \neq x,\ y = x \\ \mathsf{W}(z)(y) & \text{otherwise} \end{cases}$
$\mathsf{V}'_\mathsf{RMW} = \lambda \pi, y.$	$\begin{cases} \emptyset & \pi = \tau,\ y = x \\ \mathsf{V}_\mathsf{RMW}(\pi)(x) \cup \{v_\mathsf{R}\} & \pi \neq \tau,\ y = x \\ \mathsf{V}_\mathsf{RMW}(\pi)(y) & y \neq x \end{cases}$	$\begin{cases} \mathsf{V}_\mathsf{RMW}(\tau)(y) \cap \mathsf{W}_\mathsf{RMW}(x)(y) & \pi = \tau \\ \mathsf{V}_\mathsf{RMW}(\pi)(y) & \pi \neq \tau \end{cases}$	
$\mathsf{W}'_\mathsf{RMW} = \lambda z, y.$	$\begin{cases} \mathsf{V}_\mathsf{RMW}(\tau)(y) & z = x,\ y \neq x \\ \mathsf{W}_\mathsf{RMW}(z)(x) \cup \{v_\mathsf{R}\} & z \neq x,\ y = x \\ \mathsf{W}_\mathsf{RMW}(z)(y) & \text{otherwise} \end{cases}$	$\mathsf{W}_\mathsf{RMW}(z)(y)$	$\begin{cases} \mathsf{W}_\mathsf{RMW}(x)(y) \cap \mathsf{V}_\mathsf{RMW}(\tau)(y) & z = x,\ y \neq x \\ \mathsf{W}_\mathsf{RMW}(z)(y) & \text{otherwise} \end{cases}$

Figure 6. Maintaining V, W, V_RMW, and W_RMW in SCM transitions.

Putting all pieces together, the states of SCM are tuples $I = \langle M, V_\text{SC}, M_\text{SC}, W_\text{SC}, V, W, V_\mathsf{RMW}, M_\mathsf{RMW} \rangle$. Its transitions are obtained by instrumenting the transitions of SC (which govern the M component) with the transformations in Figures 5 and 6. The next lemma (which we proved in Coq) ensures that they track the intended properties.

Lemma 5.2. *The following hold:*

- $\mathsf{SCM.q_\emptyset} = I(G_\emptyset)$.
- *If* $G \xrightarrow{\langle \tau, l \rangle}_\text{SCG} G'$, *then* $I(G) \xrightarrow{\langle \tau, l \rangle}_\text{SCM} I(G')$.
- *If* $I(G) \xrightarrow{\langle \tau, l \rangle}_\text{SCM} I'$, *then* $G \xrightarrow{\langle \tau, l \rangle}_\text{SCG} G'$ *and* $I(G') = I'$ *for some* $G' \in \mathsf{EGraph}$.

Our main result easily follows from Thm. 5.1 and Lemma 5.2:

Theorem 5.3. *P is execution-graph robust against* RA *iff for every reachable state $\langle \overline{q}, I \rangle$ in P_SCM, the following hold for every $\langle \tau, l \rangle$ that is enabled in $\overline{q}$ and satisfies* $\mathtt{loc}(l) \in I.\mathsf{V}_\text{SC}(\tau)$, *where* $x = \mathtt{loc}(l)$ *and* $v_\mathsf{R} = \mathtt{val}_\mathsf{R}(l)$*:*

- *if* $\mathtt{typ}(l) = \mathsf{W}$ *then* $I.\mathsf{V}_\mathsf{RMW}(\tau)(x) = \emptyset$.
- *if* $\mathtt{typ}(l) = \mathsf{R}$ *then* $v_\mathsf{R} \notin I.\mathsf{V}(\tau)(x)$.
- *if* $\mathtt{typ}(l) = \mathsf{RMW}$ *then* $v_\mathsf{R} \notin I.\mathsf{V}_\mathsf{RMW}(\tau)(x)$.

PSPACE-completeness (assuming bounded data domain as we defined in §2) easily follows:

Corollary 5.4. *Verifying execution-graph robustness against* RA *for a given input program is PSPACE-complete.*

Proof (outline). For the upper bound, we can (gradually) guess a run of P_SCM and check the conditions of Thm. 5.3 at each step. The memory required for storing a state is polynomial in the size of P. The lower bound is established as the one in [19] for TSO, by a reduction from reachability under SC (which is PSPACE-complete [35]): A program can be made robust by adding fences (as in Ex. 3.6) between every two instructions, and an artificial robustness violation (e.g., in the form of SB) can be added when the target state is reached. □

Note that for verifying robustness we generate one reachability query, and since we only monitor traces, we do not add additional non-determinism w.r.t. reachability under SC. However, the instrumentation in SCM creates dependencies between instructions (e.g., both a write to x and a write to $y \neq x$ require to update the bit representing $y \in \mathsf{M}_\text{SC}(x)$), which may hinder partial order reduction.

5.1 Abstract Value Management

The V and V_RMW (and, consequently, W and W_RMW) components in SCM states are often "too elaborate" for what is actually needed to verify robustness. For example, for a program P without CAS, wait and BCAS instructions, whether P_RAG enables a transition or not does not depend on the value being read. In such case, we only need to check whether $I.\mathsf{V}(\tau)(x)$ is empty (for reads) and whether $I.\mathsf{V}_\mathsf{RMW}(\tau)(x)$ is empty (for writes and RMWs). More generally, we only need to track values that may affect P_RAG transitions (e.g., block a thread from executing or make an RMW succeed). Next, we use this observation to reduce the metadata size in SCM. To do so, we first define *critical values.*

Definition 5.5. A value $v \in \mathsf{Val}$ is called a *critical value* of $x \in \mathsf{Loc}$ in a sequential program S if at least one of the following hold for some $q \in S.\mathsf{Q}$: (1) q enables $\mathsf{R}(x, v)$ but there exists v' such that q does not enable $\mathsf{R}(x, v')$ and $\mathsf{RMW}(x, v', v_\mathsf{W})$ for every v_W; (2) q enables $\mathsf{RMW}(x, v, v_\mathsf{W})$ for some $v_\mathsf{W} \in \mathsf{Val}$ but there exists v' such that q does not enable $\mathsf{RMW}(x, v', v'_\mathsf{W})$ for every v'_W. We call v a critical value of x in a (concurrent) program P if it is a critical value of x in $P(\tau)$ for some $\tau \in \mathsf{Tid}$, and denote by $\mathsf{Val}(P, x)$ the set of critical values of x in P.

For instance, if $\mathtt{wait}(x = 1)$ is included in a program P then 1 is a critical value of x in P. Similarly, $r := \mathtt{CAS}(x, 0 \to 1)$ (e.g., for implementing spin locks) makes 0 a critical value of x. A program without CAS, wait and BCAS instructions has no critical values. On the other hand, in a program including an instruction like $r := \mathtt{CAS}(x, r' \to e)$ (where the expected value is not a constant), we have $\mathsf{Val}(P, x) = \mathsf{Val}$ (in which case, our proposed optimization does not change anything).

Now, the $\mathtt{V}, \mathtt{V}_{\mathtt{RMW}}, \mathtt{W}, \mathtt{W}_{\mathtt{RMW}}$ components can be restricted to record information only about the critical values (so, we have $\mathtt{V}, \mathtt{V}_{\mathtt{RMW}} : \mathsf{Tid} \to \prod_{x \in \mathsf{Loc}} \mathcal{P}(\mathsf{Val}(P, x))$ and $\mathtt{W}, \mathtt{W}_{\mathtt{RMW}} : \mathsf{Loc} \to \prod_{x \in \mathsf{Loc}} \mathcal{P}(\mathsf{Val}(P, x))$), and additional components $\mathtt{CV}, \mathtt{CV}_{\mathtt{RMW}} : \mathsf{Tid} \to \mathcal{P}(\mathsf{Loc})$ and $\mathtt{CW}, \mathtt{CW}_{\mathtt{RMW}} : \mathsf{Loc} \to \mathcal{P}(\mathsf{Loc})$ (disjunctively) summarize all non-critical values. The latter are formally interpreted as follows (using the interpretations above):

$$\begin{aligned}
I(G).\mathtt{CV} &= \lambda\tau.\, \{y \mid I(G).\mathtt{V}(\tau)(y) \setminus \mathsf{Val}(P, y) \neq \emptyset\} \\
I(G).\mathtt{CV}_{\mathtt{RMW}} &= \lambda\tau.\, \{y \mid I(G).\mathtt{CV}_{\mathtt{RMW}}(\tau)(y) \setminus \mathsf{Val}(P, y) \neq \emptyset\} \\
I(G).\mathtt{CW} &= \lambda x.\, \{y \mid I(G).\mathtt{W}(x)(y) \setminus \mathsf{Val}(P, y) \neq \emptyset\} \\
I(G).\mathtt{CW}_{\mathtt{RMW}} &= \lambda x.\, \{y \mid I(G).\mathtt{W}_{\mathtt{RMW}}(x)(y) \setminus \mathsf{Val}(P, y) \neq \emptyset\}
\end{aligned}$$

That is, $\mathtt{CV}(\tau)$ (respectively, $\mathtt{CV}_{\mathtt{RMW}}(\tau)$) contains all locations y for which there exist at least one non-critical value that is written by a non-mo-maximal write to y that can serve as the predecessor write in an RAG read (respectively, write or RMW) step. The maintenance of these components (given in [1, §C]) is straightforwardly derived from the maintenance of $\mathtt{V}, \mathtt{V}_{\mathtt{RMW}}, \mathtt{W}, \mathtt{W}_{\mathtt{RMW}}$.

In turn, three conditions are added to Thm. 5.3:

- if $\mathtt{typ}(l) = \mathtt{W}$ then $x \notin I.\mathtt{CV}_{\mathtt{RMW}}(\tau)$.
- if $\mathtt{typ}(l) = \mathtt{R}$ and $v_{\mathtt{R}} \notin \mathsf{Val}(P, x)$ then $x \notin I.\mathtt{CV}(\tau)$.
- if $\mathtt{typ}(l) = \mathtt{RMW}$ and $v_{\mathtt{R}} \notin \mathsf{Val}(P, x)$ then $x \notin I.\mathtt{CV}_{\mathtt{RMW}}(\tau)$.

This construction results in smaller instrumentation (and fewer operations to maintain the instrumentation), where the size (number of bits) of the monitoring metadata is

$$3|\mathsf{Tid}||\mathsf{Loc}| + 4|\mathsf{Loc}|^2 + 2(|\mathsf{Tid}| + |\mathsf{Loc}|) \sum_{x \in \mathsf{Loc}} |\mathsf{Val}(P, x)|.$$

In particular, for programs without `CAS`, `wait` and `BCAS` instructions the metadata size is $3|\mathsf{Tid}||\mathsf{Loc}| + 4|\mathsf{Loc}|^2$, while in the worst case (when all values are critical) we will have $|\mathsf{Loc}|(|\mathsf{Tid}| + 2|\mathsf{Loc}| + 2|\mathsf{Val}|(|\mathsf{Tid}| + |\mathsf{Loc}|))$. In some of the examples we checked, this optimization dramatically reduce the verification time (e.g., the 'ticketlock4' example in §7 is x9 faster). In addition, it may be beneficial for programs with infinite data domains but finite sets of critical values, where the (generally undecidable) reachability problem in P_{SCM} can be solved using abstraction techniques. (This is left for future work.)

6 Extension with Non-atomic Accesses

In this section, we describe an extension of our approach to handle C/C++11's non-atomic accesses, typically used for "data variables" (unlike "synchronization variables"). A data-race on a non-atomic access is considered an undefined behavior, and thus non-atomic accesses allow very efficient implementation. In turn, robustness of a program should imply that it has no data-races on non-atomic accesses.

For this extension, we assume that $\mathsf{Loc} = \mathsf{Loc}_{\mathsf{ra}} \uplus \mathsf{Loc}_{\mathsf{na}}$ is composed from a set of *release/acquire* locations and a disjoint set of *non-atomic* locations (we do not consider release/acquire and non-atomic accesses to the same location). The programming language Fig. 1 is extended with instructions $x_{\mathsf{na}} := e$ and $r := x_{\mathsf{na}}$ for $x_{\mathsf{na}} \in \mathsf{Loc}_{\mathsf{na}}$, $e \in \mathsf{Exp}$, and $r \in \mathsf{Reg}$. The rest of the instructions only apply to locations in $\mathsf{Loc}_{\mathsf{ra}}$ (in particular, there are no RMW instructions for non-atomic locations).

The SC and SCG systems ignore the type of the location, while RAG is extended to detect races on non-atomic locations. We refer to the extended memory subsystem as RAG+NA. The state of RAG+NA are execution graphs (as in RAG) as well as a special state, denoted by $\bot$, that the system enters once a race is detected. To define RAG+NA's transitions, hb is modified so that only $\mathtt{rf}$-edges on release/acquire accesses synchronize:

$$G.\mathsf{hb} \triangleq (G.\mathsf{po} \cup \bigcup_{x \in \mathsf{Loc}_{\mathsf{ra}}} [\mathsf{W}_x] \,;\, G.\mathtt{rf} \,;\, [\mathsf{R}_x])^+$$

Now, the transitions of RAG+NA extend the transitions of RAG (which govern the release/acquire locations) with the following steps for non-atomic accesses:

$$\frac{\begin{array}{c} x_{\mathsf{na}} = \mathtt{loc}(l) \qquad x_{\mathsf{na}} \in \mathsf{Loc}_{\mathsf{na}} \\ \mathtt{typ}(l) = \mathtt{R} \implies \mathtt{val}_{\mathtt{W}}(G.w^{\max}_{x_{\mathsf{na}}}) = \mathtt{val}_{\mathtt{R}}(l) \\ G.w^{\max}_{x_{\mathsf{na}}} \in \mathit{dom}(G.\mathsf{hb}^? \,;\, [G.\mathsf{E}^\tau]) \end{array}}{G \xrightarrow{\langle\tau,l\rangle}_{\mathsf{RAG+NA}} \mathsf{add}(G, \tau, l, G.w^{\max}_x)}$$

$$\frac{\mathtt{loc}(l) \in \mathsf{Loc}_{\mathsf{na}} \qquad G.w^{\max}_{\mathtt{loc}(l)} \notin \mathit{dom}(G.\mathsf{hb}^? \,;\, [G.\mathsf{E}^\tau])}{G \xrightarrow{\langle\tau,l\rangle}_{\mathsf{RAG+NA}} \bot}$$

Thus, for a thread to successfully perform a non-atomic access to location x_{na}, it must have observed (in hb) the mo-maximal (equivalently, hb-maximal) write to x_{na}. Otherwise, the system moves to the $\bot$ state.

Execution-graph robustness against RAG+NA is defined just as against RA (cf. Def. 4.9), and it implies state robustness against RAG+NA. Since P_{SCG} never reaches states of the form $\langle\overline{q}, \bot\rangle$, execution-graph robustness against RAG+NA implies that such states are not reachable in $P_{\mathsf{RAG+NA}}$. Next, Theorem 5.1 is extended as follows:

Definition 6.1. A state $\overline{q}$ of a concurrent program is *racy* if $\overline{q}$ enables both $\langle\tau, l_1\rangle$ and $\langle\pi, l_2\rangle$ for some $\tau \neq \pi$ and $l_1, l_2 \in \mathsf{Lab}$ with $\mathtt{loc}(l_1) = \mathtt{loc}(l_2) \in \mathsf{Loc}_{\mathsf{na}}$ and $\mathtt{W} \in \{\mathtt{typ}(l_1), \mathtt{typ}(l_2)\}$.

Theorem 6.2. *A concurrent program P is execution-graph robust against* RAG+NA *iff there does not exist a non-robustness witness $\langle\overline{q}, G, \tau, l, w\rangle$ for P with $\mathtt{loc}(l) \in \mathsf{Loc}_{\mathsf{ra}}$ (as defined in Thm. 5.1), and there does not exist a reachable state $\langle\overline{q}, G\rangle$ in P_{SCG} such that $\overline{q}$ is racy.*

The SCM system can be easily adapted for monitoring the conditions of Thm. 6.2. The memory component in SCM's

Program	Res	#T	LoC	Time	SC	Trencher TSO	
						Res	Time
barrier (BAR)	✓	2	11	1.6 (100%)	1.1	✗⋆	-
dekker-sc	✗	2	43	4.2 (100%)	1.3	✗	5.9
dekker-tso	✓	2	49	5.2 (100%)	1.3	✓	5.9
peterson-sc	✗	2	28	2.5 (100%)	1.2	✗	5.6
peterson-tso	✗	2	30	3.3 (100%)	1.3	✓	5.6
peterson-ra	✓	2	44	5.8 (100%)	1.2	✓	5.8
peterson-ra-dmitriy	✓	2	36	4.3 (100%)	1.2	✓	5.5
peterson-ra-bratosz	✗	2	28	3.4 (100%)	1.1	✗	5.6
lamport2-sc	✗	2	65	9.1 (100%)	1.3	✗	8.0
lamport2-tso	✗	2	69	13.7 (100%)	1.3	✓	8.2
lamport2-ra	✓	2	79	18.9 (99%)	1.4	✓	7.8
lamport2-3-ra	✓	3	123	215.6 (21%)	6.1	✗⋆	-
spinlock	✓	2	34	1.6 (100%)	1.2	✓	5.4
spinlock4	✓	4	66	6.4 (80%)	1.6	✓	6.8
ticketlock	✓	2	25	2.6 (100%)	1.1	✓	5.8
ticketlock4	✓	4	49	22.6 (25%)	7.5	✓	23.4
seqlock	✓	4	49	20.7 (16%)	3.4	✓	8.9
nbw-w-lr-rl	✓	4	50	5.7 (100%)	1.2	✓	8.6
rcu	✓	4	74	67.6 (10%)	2.2	✗⋆	-
rcu-offline	✓	3	215	137.9 (50%)	18.3	✗⋆	-
cilk-the-wsq-sc	✗	2	57	5.0 (100%)	1.2	✗	9.6
cilk-the-wsq-tso	✓	2	59	6.1 (100%)	1.3	✓	11.7
chase-lev-sc	✗	3	55	3.8 (100%)	29.5	✗	15.3
chase-lev-tso	✗	3	57	4.9 (100%)	31.3	✓	128.1
chase-lev-ra	✓	3	61	67.1 (8%)	38.1	✓	108.3

Figure 7. Experiments with *Rocker*

states is extended in the obvious way to track the latest value of non-atomic locations as well. Since non-atomic instructions do not affect inter-thread synchronization, the monitoring instrumentation in §5 requires no change (it only applies to the locations in $\mathsf{Loc}_{\mathsf{ra}}$). Since SCM and SCG have the same traces, the additional condition about races can be checked on SCM runs.

7 Implementation and Evaluation

We implemented our algorithm in a prototype tool called *Rocker* (for RObustness ChecKER), which uses Spin [31] as a back-end model checker. The implementation and the examples it was tested on are available in the artifact accompanying this paper. *Rocker* takes as input a program in our toy programming language, and converts it to Promela code (Spin's input language) with appropriate instrumentation and assertions that check for execution-graph robustness against RA. Thus, our implementation is actually using the SC memory subsystem, and implements the monitoring of SCM by instrumenting the input program. When a robustness violation is detected, one can use Spin's output to see the trace leading to this violation. In addition, since in any case we explore traces of the input program under SC, *Rocker* allows one to include standard assertions, which will be verified as well by the model checker.

We performed a series of experiments on litmus tests, examples from [5, 17], and additional concurrent algorithms. Figure 7 summarizes the running times on some of the examples when executed on an Intel® Core™ i5-6300U CPU @ 2.40GHz GNU/Linux machine. Columns 'Res', '#T', and 'LoC' respectively present the robustness of the input program, the number of threads, and total number of lines of code. Column 'Time' shows the verification time (in seconds), and the percentage of that time that was dedicated to compiling Spin's verifier (using gcc with -O2). The latter often completely dominates the total time. Generating the input for Spin is negligibly fast ($<0.1s$), as well as Spin's verifier generation in C ($<0.2s$). Column 'SC' provides, for the sake of comparison, the verification duration using Spin with no instrumentation whatsoever. In this mode, only the assertions in the input are verified assuming SC semantics.

For some of the examples Fig. 7 provides several versions of the same algorithm: The '-sc' suffix denotes an original algorithm as designed for SC; the '-tso' suffix denotes its strengthening with fences to ensure robustness against TSO; and, when needed, the '-ra' suffix is a further strengthening that ensures robustness against RA. For instance, it is well known that Peterson mutual exclusion algorithm ('peterson-sc') is not robust against relaxed memory. For TSO, placing one fence in each thread suffices to ensure robustness. For RA, more fences are needed ('peterson-ra'). Alternatively, as noted in [57], one may replace certain write operations by RMWs ('peterson-dmitriy'). The choice of these writes is critical—*Rocker* correctly identified that a different version is incorrect ('peterson-bratosz'). Other algorithms, which were designed with relaxed memory considerations in mind, e.g., Seqlocks [16] and a user-level RCU [26], do not require fences at all. Note that we have also verified robustness of a more involved RCU implementation ('rcu-offline'), where the writer is not a unique thread, and threads may declare that they are going offline, stop the communication with the writer and return online later on.

Finally, column 'Trencher' provides the (total) running time of Trencher, a tool for verifying robustness against TSO [17], which also uses Spin for model checking. (A newer version of Trencher that implements its own model checker crashed on some of these examples.) Their notion of robustness is similar to execution-graph robustness, but it should be noted that *Rocker* and Trencher solve different problems: TSO and RA are fundamentally different models, where RA is weaker and non-multi-copy atomic. Thus, this comparison is of limited significance (see also §8). The input language is different as well. In particular, Trencher does not handle blocking instructions. For this reason, Trencher reports some examples as non-robust (marked with ⋆), while no additional fences are needed for them to function correctly under TSO. We note that Trencher can be used in parallel to *Rocker* for verifying robustness against RA: a violation detected by Trencher implies non-robustness against RA.

8 Related Work

Robustness against weak memory semantics was studied before for *hardware* models, especially in the context of automatically enforcing robustness by inserting memory fences and other synchronization primitives (see, e.g., [9, 21, 24, 25], as well as [8] for a practical approximate generic approach).

In particular, robustness against TSO (and its PSO variant) [10, 32, 50] received considerable attention, e.g., [4, 5, 17–19, 22, 23, 30, 41–43, 49]. Generally speaking, the closest to our approach is Burckhardt and Musuvathi [22], implemented in a tool called Sober, which reduces robustness against TSO to reachability under SC in an instrumented program that verifies that TSO executions cannot diverge w.r.t. SC ones.[3] In addition, verifying (trace based) robustness against TSO was shown by Bouajjani et al. [19] to be PSPACE complete—the same complexity, as we show, as verifying execution-graph robustness against RA.

Except for the fact that RA is strictly weaker than TSO (see the 2+2W and IRIW programs above), there are crucial differences between TSO and RA that do not allow one to apply the approaches developed for TSO when targeting RA. First, TSO's operational model provides a simple description of its runs, identifying a TSO run with an SC run where global effects of write instructions may be delayed. This presentation of TSO plays a key role in the characterization, verification and enforcement of robustness against TSO (see, e.g., [5, 17–19]). RA does not admit a similar presentation, and in fact, since RA is non-multi-copy-atomic (see Ex. 3.3), unlike TSO, RA cannot be explained by program transformations (instruction reorderings and eliminations) on top of SC [38]. Second, RMW operations in RA provide much weaker guarantees than in TSO, where even a failed CAS (when a CAS instruction is included as a primitive, as in [43]) serves as a memory fence. As described in §5, handling RMWs in RA (where, in particular, a failed CAS is nothing more than a plain read) requires certain technical novelties.

Less work was devoted to robustness against a *programming language* concurrency semantics. The well-known DRF guarantee [6, 29] is a simple robustness criterion, e.g., for a strengthened version of C11 [13, 39], but it is too weak, as (low-level) synchronizations naturally involve data-races, and often do not imply non-robustness. Meshman et al. [46] proposed an (approximate and incomplete) method that uses CDSchecker [48] for restricting non-SC behaviors of C11 programs. For a particular class of "server client programs", it was shown in [36] that certain simple fence insertion strategy ensures robustness. However, in this paper we are interested in precise robustness verification for arbitrary programs.

Verification under RA has also received significant attention. This includes works on program logics, e.g., [27, 33, 37, 53–55], which require manual proofs, and (bounded) model checkers, e.g., [3, 34, 48], which provide limited guarantees for programs with loops. These methods can be used to verify programs that are not necessarily robust against RA. The verification problem of programs with loops under RA (i.e., given a program P and a state $\overline{q} \in P.\mathrm{Q}$, is $\overline{q}$ reachable under the concurrent system P_{RA}?) was recently shown to be undecidable [2]. (For TSO, this problem is decidable but non-primitive recursive [11, 12].) As shown in [24, Thm. 2.12], this immediately entails the undecidability of state robustness.

Finally, robustness was also studied, e.g., in [15, 20, 28, 47], in the context of distributed systems, where SC is replaced by *serializability*. Unlike the current work, these works are focused on practical over-approximations, and do not provide provably precise general verification methods.

9 Conclusion

We have presented a method to verify execution-graph robustness against release/acquire concurrency semantics, in particular, establishing the decidability of this problem. Our method works by exploring only runs of the program under SC while monitoring certain properties for the detection of robustness violations. We believe that our result can play an important role in verification and development of concurrent algorithms for weak memory semantics, alongside with other existing methods.

In the future, we plan to study the applicability of our approach for different and extended models, such as RC11 [39], WeakRC11 [34], SRA [36], as well as transactional consistency models, such as PSI [52]. In addition, we are interested in deriving efficient and precise methods for automatic robustness enforcement (such as fence insertion) as were developed before for hardware models; as well as in handling parametrized programs with arbitrary number of threads.

Acknowledgments

We thank the PLDI'19 reviewers for their helpful feedback. This research was supported by the Israel Science Foundation (grant number 5166651), 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 Roy Margalit. 2019. Supplementary material for this paper. https://www.cs.tau.ac.il/~orilahav/papers/pldi19full.pdf

[2] Parosh Aziz Abdulla, Jatin Arora, Mohamed Faouzi Atig, and Shankaranarayanan Krishna. 2019. Verification of programs under the release-acquire semantics. In *PLDI (to appear)*.

[3] 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

[4] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Magnus Lång, and Tuan Phong Ngo. 2015. Precise and sound automatic fence insertion

[3] However, as Burnim et al. [23] observed (and as was verified in [44]), the declarative TSO model in [22] is broken (it mishandles internal reads-from edges), rendering Sober unsound.

procedure under PSO. In *NETYS*. Springer International Publishing, Cham, 32–47.

[5] Parosh Aziz Abdulla, Mohamed Faouzi Atig, and Tuan-Phong Ngo. 2015. The best of both worlds: Trading efficiency and optimality in fence insertion for TSO. In *ESOP*. Springer-Verlag New York, Inc., New York, 308–332. https://doi.org/10.1007/978-3-662-46669-8_13

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

[7] Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli, and Phillip W. Hutto. 1995. Causal memory: definitions, implementation, and programming. *Distributed Computing* 9, 1 (1995), 37–49.

[8] Jade Alglave, Daniel Kroening, Vincent Nimal, and Daniel Poetzl. 2017. Don't sit on the fence: a static analysis approach to automatic fence insertion. *ACM Trans. Program. Lang. Syst.* 39, 2, Article 6 (May 2017), 38 pages. https://doi.org/10.1145/2994593

[9] Jade Alglave and Luc Maranget. 2011. Stability in weak memory models. In *CAV*. Springer-Verlag, Berlin, Heidelberg, 50–66. http://dl.acm.org/citation.cfm?id=2032305.2032311

[10] 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

[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, 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, 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] Hans-J. Boehm. 2012. Can Seqlocks get along with programming language memory models?. In *MSPC*. ACM, New York, 12–20. https://doi.org/10.1145/2247684.2247688

[17] Ahmed Bouajjani, Egor Derevenetc, and Roland Meyer. 2013. Checking and enforcing robustness against TSO. In *ESOP*. Springer-Verlag, Berlin, Heidelberg, 533–553. https://doi.org/10.1007/978-3-642-37036-6_29

[18] Ahmed Bouajjani, Constantin Enea, Suha Orhun Mutluergil, and Serdar Tasiran. 2018. Reasoning about TSO programs using reduction and abstraction. In *CAV*. Springer, Cham, 336–353.

[19] Ahmed Bouajjani, Roland Meyer, and Eike Möhlmann. 2011. Deciding robustness against total store ordering. In *ICALP*. Springer, Berlin, Heidelberg, 428–440.

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

[21] Sebastian Burckhardt, Rajeev Alur, and Milo M. K. Martin. 2007. CheckFence: Checking consistency of concurrent data types on relaxed memory models. In *PLDI*. ACM, New York, 12–21. https://doi.org/10.1145/1250734.1250737

[22] Sebastian Burckhardt and Madanlal Musuvathi. 2008. Effective program verification for relaxed memory models. In *CAV*. Springer-Verlag, Berlin, Heidelberg, 107–120. https://doi.org/10.1007/978-3-540-70545-1_12

[23] Jabob Burnim, Koushik Sen, and Christos Stergiou. 2011. Sound and complete monitoring of sequential consistency for relaxed memory models. In *TACAS*. Springer, Berlin, Heidelberg, 11–25.

[24] Egor Derevenetc. 2015. *Robustness against relaxed memory models*. Ph.D. Dissertation. University of Kaiserslautern. http://kluedo.ub.uni-kl.de/frontdoor/index/index/docId/4074

[25] Egor Derevenetc and Roland Meyer. 2014. Robustness against Power is PSpace-complete. In *ICALP*. Springer, Berlin, Heidelberg, 158–170.

[26] Mathieu Desnoyers, Paul E. McKenney, Alan S. Stern, Michel R. Dagenais, and Jonathan Walpole. 2012. User-level implementations of read-copy update. *IEEE Trans. Parallel Distrib. Syst.* 23, 2 (Feb. 2012), 375–382. https://doi.org/10.1109/TPDS.2011.159

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

[28] Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and Dennis Shasha. 2005. Making snapshot isolation serializable. *ACM Trans. Database Syst.* 30, 2 (June 2005), 492–528. https://doi.org/10.1145/1071610.1071615

[29] Kourosh Gharachorloo, Sarita V. Adve, Anoop Gupta, John L. Hennessy, and Mark D. Hill. 1992. Programming for different memory consistency models. *J. Parallel and Distrib. Comput.* 15, 4 (1992), 399 – 407. https://doi.org/10.1016/0743-7315(92)90052-O

[30] Alexey Gotsman, Madanlal Musuvathi, and Hongseok Yang. 2012. Show no weakness: sequentially consistent specifications of TSO libraries. In *DISC*. Springer-Verlag, Berlin, Heidelberg, 31–45. https://doi.org/10.1007/978-3-642-33651-5_3

[31] Gerard J. Holzmann. 1997. The model checker SPIN. *IEEE Transactions on software engineering* 23, 5 (1997), 279–295.

[32] SPARC International Inc. 1994. *The SPARC architecture manual (version 9)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[33] 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

[34] 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

[35] Dexter Kozen. 1977. Lower bounds for natural proof systems. In *SFCS*. IEEE Computer Society, Washington, 254–266. https://doi.org/10.1109/SFCS.1977.16

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

[37] 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

[38] 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

[39] 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, 618–632. https://doi.org/10.1145/3062341.3062352

[40] Leslie Lamport. 1979. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers* 28, 9 (1979), 690–691.

[41] Alexander Linden and Pierre Wolper. 2011. A verification-based approach to memory fence insertion in relaxed memory systems. In *SPIN*. Springer-Verlag, Berlin, Heidelberg, 144–160. http://dl.acm.org/citation.cfm?id=2032692.2032707

[42] Alexander Linden and Pierre Wolper. 2013. A verification-based approach to memory fence insertion in PSO memory systems. In *TACAS*. Springer-Verlag, Berlin, Heidelberg, 339–353. https://doi.org/10.1007/

978-3-642-36742-7_24

[43] Feng Liu, Nayden Nedev, Nedyalko Prisadnikov, Martin Vechev, and Eran Yahav. 2012. Dynamic synthesis for relaxed memory models. In *PLDI*. ACM, New York, 429–440. https://doi.org/10.1145/2254064.2254115

[44] Sela Mador-Haim, Rajeev Alur, and Milo M K. Martin. 2010. Generating litmus tests for contrasting memory consistency models. In *CAV*. Springer-Verlag, Berlin, Heidelberg, 273–287. https://doi.org/10.1007/978-3-642-14295-6_26

[45] 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.

[46] Yuri Meshman, Noam Rinetzky, and Eran Yahav. 2015. Pattern-based synthesis of synchronization for the C++ memory model. In *FMCAD*. FMCAD Inc, Austin, TX, 120–127. http://dl.acm.org/citation.cfm?id=2893529.2893552

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

[48] Brian Norris and Brian Demsky. 2013. CDSchecker: checking concurrent data structures written with C/C++ atomics. In *OOPSLA*. ACM, New York, 131–150. https://doi.org/10.1145/2509136.2509514

[49] Scott Owens. 2010. Reasoning about the implementation of concurrency abstractions on x86-TSO. In *ECOOP*. Springer-Verlag, Berlin, Heidelberg, 478–503.

[50] 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

[51] Dennis Shasha and Marc Snir. 1988. Efficient and correct execution of parallel programs that share memory. *ACM Trans. Program. Lang. Syst.* 10, 2 (April 1988), 282–312. https://doi.org/10.1145/42190.42277

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

[53] Kasper Svendsen, Jean Pichon-Pharabod, Marko Doko, Ori Lahav, and Viktor Vafeiadis. 2018. A separation logic for a promising semantics. In *ESOP*. Springer International Publishing, Cham, 357–384.

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

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

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

[57] Anthony Williams. 2008. Peterson's lock with C++0x atomics. Retrieved October 26, 2018 from https://www.justsoftwaresolutions.co.uk/threading/petersons_lock_with_C++0x_atomics.html