Verifying Observational Robustness against a C11-Style Memory Model

ROY MARGALIT, Tel Aviv University, Israel
ORI LAHAV, Tel Aviv University, Israel

We study the problem of verifying the robustness of concurrent programs against a C11-style memory model that includes relaxed accesses and release/acquire accesses and fences, and show that this verification problem can be reduced to a standard reachability problem under sequential consistency. We further observe that existing robustness notions do not allow the verification of programs that use speculative reads as in the sequence lock mechanism, and introduce a novel "observational robustness" property that fills this gap. In turn, we show how to soundly check for observational robustness. We have implemented our method and applied it to several challenging concurrent algorithms, demonstrating the applicability of our approach. To the best of our knowledge, this is the first method for verifying robustness against a programming language concurrency model that includes relaxed accesses and release/acquire fences.

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

Additional Key Words and Phrases: weak memory models, C/C++11, robustness, shared-memory concurrency

ACM Reference Format:
Roy Margalit and Ori Lahav. 2021. Verifying Observational Robustness against a C11-Style Memory Model. *Proc. ACM Program. Lang.* 5, POPL, Article 4 (January 2021), 33 pages. https://doi.org/10.1145/3434285

1 INTRODUCTION

Programming and reasoning about concurrent programs under memory models weaker than sequential consistency (SC) is nutritiously hard and error-prone. When programmers manage to avoid data races, the DRF (data-race-freedom) guarantee allows them to safely imagine a heaven of SC multiprocessors and SC-preserving compilers [Adve and Hill 1990]. For a variety of programs, however, avoiding races altogether is too restrictive and costly. Interestingly, often this does not mean that the program may behave under a weak memory model differently than it would behave under SC. Accordingly, there is a need to develop *robustness* criteria (i.e., conditions that ensure that a given program has only SC behaviors), which are more precise than DRF, and accompany them with robustness verification methods and tools. Safety verification under a weak memory model can be then reduced to robustness verification plus safety verification assuming SC [Burckhardt and Musuvathi 2008].

Precise robustness criteria and robustness verification tools were previously developed for hardware models (x86-TSO, see, e.g., [Bouajjani et al. 2013, 2011; Owens 2010], and POWER [Derevenetc and Meyer 2014]), and recently for the release/acquire fragment (RA) of C11 [Lahav and Margalit 2019]. In this work we are interested in a larger fragment of C11, which also includes relaxed

Authors' addresses: Roy Margalit, Tel Aviv University, Israel, roy.margalit@cs.tau.ac.il; Ori Lahav, Tel Aviv University, Israel, orilahav@tau.ac.il.

2475-1421/2021/1-ART4
https://doi.org/10.1145/3434285

accesses and release/acquire fences. These provide more fine-grained control on the required synchronization, which allows for better performance. The price to pay is that we get much farther than SC: relaxed accesses do not maintain basic causality as they can be reordered[1] (see, e.g., Ex. 4.2 below); release and acquire fences separate the synchronization points from the actual writes and reads [Batty et al. 2011]; and *release sequences*, a special mechanism for using relaxed read-modify-write instructions (RMWs) for synchronization, are hard to reason about [Doko and Vafeiadis 2017]. This makes programming more difficult and error-prone (see, e.g., the case of Peterson's lock described in §7), which, in fact, may (and, we believe, often does) cause programmers to refrain from using relaxed accesses altogether.

In this work, we extend the results of Lahav and Margalit [2019] to a larger fragment of C11, which also includes relaxed reads/writes/RMWs and release/acquire fences. We show that robustness against this model is reduced to a reachability problem under an instrumented SC semantics.[2] Roughly speaking, while the generated execution histories (a.k.a. *execution graphs*) are unbounded (since programs may have loops), we show that a finite collection of (rather intricate) properties of the generated execution history suffices for (sound and precise) monitoring of states where threads may observe (either by reading or by overwriting) values under the weak semantics that cannot be observed under SC. Based on this reduction, we obtain a decision procedure for robustness, which we have also implemented and experimented with, using SPIN [Holzmann 1997] for the reachability analysis under SC. In particular, it follows that for programs with bounded data domain robustness verification against the model studied here is PSPACE-complete (just like robustness verification against x86-TSO or RA).

The factors that make programming with relaxed accesses to be difficult also make robustness verification for relaxed accesses to be technically challenging. Indeed, the instrumented semantics that we develop requires much finer distinctions maintaining multiple views for each thread, which in turn allow us to compare the observable values under SC to those observable in the relaxed semantics. Nevertheless, while the instrumentation is heavier than the one in [Lahav and Margalit 2019], our experiments show that this has only a minor effect on the practical verification times.

While handling relaxed accesses, we have identified a drawback of all (to the best of our knowledge) existing robustness criteria, which becomes more important for robustness against a semantics that includes relaxed accesses. It is related to the question of what exactly constitutes a program behavior (and, in turn, what is a robustness violation?). The simplest answer identifies program behaviors with sets of reachable program states, where program states consist of the values of the different program counters and local variables. However, Derevenetc [2015] observed that, while desirable, solving the robustness verification problem for a state-based robustness notion is as hard as full-fledged verification under the weak semantics. The latter can be extremely difficult from a computational complexity point of view (non-primitive recursive for TSO [Atig et al. 2010, 2012] and undecidable for RA [Abdulla et al. 2019]). Thus, researchers have resorted to stronger robustness definitions, in which program behaviors are taken to be sets of "annotated" histories of the executions of the programs (called *traces* or *execution graphs*), and robustness requires that all such histories generated by the program under the weak semantics can be also generated under SC.

However, we observe here that both state-based and history-based robustness notions forbid the benign programming idiom of *speculative* reads as used in sequence lock algorithms. These algorithms used in the Linux kernel were identified as a main use of relaxed accesses [Sinclair

[1]Like all verification works, we are unable to work with the original C11 model, which allows unrestricted cycles in the union of "program order" and "reads-from", and thus exhibits "out-of-thin-air" values and even fails to provide the most basic DRF guarantee. We follow [Boehm and Demsky 2014; Lahav et al. 2017] and disallow such cycles, which invalidates store-after-load reordering. Section 3 discusses further technical differences between the model used here and C11.
[2]Equivalently, as we do in our implementation, one can think about reachability under SC of an instrumented program.

et al. 2017], and considered as probably "the most challenging of common programming idioms to implement with modern programming language memory models" [Boehm 2012]. Roughly speaking, they are locking mechanisms in which relaxed accesses are used to efficiently speculatively read some shared data, requiring a validation mechanism (implemented with stronger accesses or fences) before the read values are being used. If validation fails, the speculations are disposed and retried. Observing stale speculative values under relaxed semantics (which cannot be observed under SC) may be perfectly fine, but it does imply a reachable intermediate program state that is not reachable under SC, resulting in a robustness violation. Nevertheless, if the validation mechanism can be proved to throw away stale values before they are actually being used, this mechanism should be considered as a benign temporary robustness violation that can never harm the program's safety.

To support verification of such mechanisms, we propose a novel notion of robustness, which we call *observational robustness*. It is a history-based notion (to make is computationally attainable), but it allows the program to have some non-SC histories provided that they can be "turned" into SC histories without affecting any value that the program used. To do so, we include a dependency relation (akin to the one used in modern hardware models, see, e.g., [Flur et al. 2016; Podkopaev et al. 2019]) in the program's execution graphs, and allow observationally robust programs to read stale values (violating SC) as long as no further operation depends on these values.

As we did for usual robustness, we show that observational robustness can be verified by inspecting *only* SC executions of the program, which, again, allows the reduction to reachability in an instrumented SC semantics. These results require certain technical novelties as we have to carefully generate an SC execution from any execution representing an observational robustness violation (that could have diverged from SC early on), as well as to devise a specialized taint tracking mechanism for identifying dependencies on stale values. We note that our observational robustness verification is sound but, unlike our result for usual (history-based) robustness, imprecise. Still, we show that it verifies the challenging examples of [Boehm 2012] deeming them as robust and safe. We leave the problem of developing a more precise analysis (as well as identifying the robustness notion that our analysis is precise for) to future work.

To conclude, the main contributions of this paper are summarized as follows:

(1) We develop a sound and precise robustness verification method (by reduction to reachability in an instrumented SC semantics) for a C11-style model that includes relaxed accesses and release/acquire fences.
(2) We introduce a notion of *observational robustness* that allows one to reason about sequence-locks-like programs using speculative reads, and provide a sound verification method for this robustness notion.
(3) We provide an implementation of our approach and present its application on multiple challenging examples.

Outline. The rest of this paper is organized as follows. In §2 we present the programming language that we assume in this paper. In §3 we present the C11-style memory model that we study in this paper and discuss its relation to (R)C11. In §4 we present our robustness notions and the key theorems that enable their verification. In §5 we present the reduction of robustness verification to a reachability problem under SC. In §7 we discuss the implementation and our experiments with it. In §8 we present related work and conclude.

Additional Material. Proofs of the theorems in the paper are given in the its accompanying technical appendix available at [Margalit and Lahav 2020]. The prototype implementation and the examples it was tested on are available in the artifact accompanying this paper (available at https://dl.acm.org/do/10.1145/3410267/full/).

$v, u \in \mathsf{Val}$	Values	$o_{\mathsf{X}} \in \mathsf{Mod}_{\mathsf{X}}$ where $\mathsf{X} \in \{\mathsf{R}, \mathsf{W}, \mathsf{RMW}, \mathsf{F}\}$	Access modes
$x, y, z \in \mathsf{Loc} \subseteq \{\mathsf{x}, \mathsf{y}, \ldots\}$	Locations	$sPr \in \mathsf{SProg} \triangleq \{0, 1, \ldots, N\} \to \mathsf{Inst}$	Sequential programs
$r \in \mathsf{Reg} \subseteq \{\mathsf{a}, \mathsf{b}, \ldots\}$	Registers	$Pr : \mathsf{Tid} \to \mathsf{SProg}$	(Concurrent) programs
$\tau, \pi \in \mathsf{Tid} \subseteq \{\mathsf{T}_1, \mathsf{T}_2, \ldots\}$	Thread identifiers		

$$
\begin{array}{rl}
e ::= & r \mid v \mid e + e \mid e = e \mid e \neq e \mid \ldots \\
\mathsf{Inst} \ni inst ::= & r := e \mid \texttt{if } e \texttt{ goto } pc_1, \ldots, pc_n \mid \texttt{assert}(e) \\
\mid & x.\texttt{store}(e, o_{\mathsf{W}}) \mid r := x.\texttt{load}(o_{\mathsf{R}}) \\
\mid & r := x.\texttt{FADD}(e, o_{\mathsf{RMW}}) \mid r := x.\texttt{CAS}(e, e, o_{\mathsf{RMW}}, o_{\mathsf{R}}) \mid \texttt{fence}(o_{\mathsf{F}}) \\
\mid & \texttt{wait}(x = e, o_{\mathsf{R}}) \mid x.\texttt{BCAS}(e, e, o_{\mathsf{RMW}})
\end{array}
$$

Fig. 1. Domains and programming language syntax.

2 CONCURRENT PROGRAMMING LANGUAGE

In this section we formulate the syntax of our C-style toy programming language (§2.1), and present the interpretation of concurrent programs as labeled transition systems synchronized with general memory systems (§2.2). We start with several preliminary conventions and notations.

Relations. 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, $[A] ; R ; [B] = R \cap (A \times B)$.

Labeled Transition Systems. A *labeled transition system* (LTS) A is a tuple $\langle Q, \Sigma, q_0, \to \rangle$, where Q is a set of *states*, Σ is an *alphabet*, $q_0 \in Q$ 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 $A.\mathsf{Q}$, $A.\Sigma$, $A.\mathsf{q}_0$, and $\to_A$ the 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 $\sigma \in A.\Sigma$ is *enabled* in $q \in A.\mathsf{Q}$ (alternatively, q *enables* σ) if $q \xrightarrow{\sigma}_A q'$ for some $q' \in A.\mathsf{Q}$. For a finite sequence $tr \in A.\Sigma^*$ (i.e., a function from $\{1, \ldots, N\}$ to $A.\Sigma$ for some $N \geq 0$), we write $\xrightarrow{tr}_A$ for the composition $\xrightarrow{tr(1)}_A ; \ldots ; \xrightarrow{tr(|tr|)}_A$. A sequence $tr \in A.\Sigma^*$ such that $A.\mathsf{q}_0 \xrightarrow{tr}_A q$ for some $q \in A.\mathsf{Q}$ is called a *trace* of A.

2.1 Language Syntax

We assume *finite* sets of values, (shared) memory locations, register names, and thread identifiers: $\mathsf{Val} \subseteq \mathbb{N}$, $\mathsf{Loc} \subseteq \{\mathsf{x}, \mathsf{y}, \ldots\}$, $\mathsf{Reg} \subseteq \{\mathsf{a}, \mathsf{b}, \ldots\}$, and $\mathsf{Tid} \subseteq \{\mathsf{T}_1, \mathsf{T}_2, \ldots\}$. We assume that Val contains a distinguished value 0, used as the initial value for all locations. Memory instructions in our language employ *access modes*, taken from the set $\mathsf{Mod} \triangleq \{\texttt{rlx}, \texttt{rel}, \texttt{acq}, \texttt{acqrel}\}$, which determine their "consistency level".[3] Its following subsets provide the possible read, write, RMW, and fence modes respectively:

$$
\begin{array}{ll}
\mathsf{Mod}_{\mathsf{R}} \triangleq \{\texttt{rlx}, \texttt{acq}\} & \qquad \mathsf{Mod}_{\mathsf{RMW}} \triangleq \{\texttt{rlx}, \texttt{acq}, \texttt{rel}, \texttt{acqrel}\} \\
\mathsf{Mod}_{\mathsf{W}} \triangleq \{\texttt{rlx}, \texttt{rel}\} & \qquad \mathsf{Mod}_{\mathsf{F}} \triangleq \{\texttt{acq}, \texttt{rel}, \texttt{acqrel}\}
\end{array}
$$

A partial order $\sqsubseteq$ on Mod, which intuitively orders access modes according to their strength, is defined by (the reflexive transitive closure of):

$$\texttt{rlx} \sqsubseteq \texttt{acq} \sqsubseteq \texttt{acqrel} \qquad \text{and} \qquad \texttt{rlx} \sqsubseteq \texttt{rel} \sqsubseteq \texttt{acqrel}$$

[3]In §6 we extend our results to support C11-style *non-atomic accesses* with na access mode.

Figure 1 presents our toy programming language. Its expressions are constructed from registers (local variables) and values. Instructions include assignments, memory accesses and release/acquire fences. SC-fences are not taken as primitives in this language but rather included as a syntactic sugar; see §3.1 and Remark 1. We include non-deterministic conditional jumps instructions ($\texttt{if}\ e\ \texttt{goto}\ pc_1, \ldots, pc_n$ jumps to some program counter pc_k if e is not 0) from which usual conditionals and loops can be implemented. Assertions can be used for safety verification ($\texttt{assert}(e)$ fails if e is 0). Memory accesses consist of loads, stores, and RMWs, which are either (never failing) fetch-and-adds (FADD) or (possibly failing) compare-and-swaps (CAS). Both RMW operations return the value that was read before its update. The language also includes two kinds of blocking instructions: $\texttt{wait}(x = e, o_{\mathsf{R}})$ blocks the execution of the thread until it can read the value of e from x (with mode $o_{\mathsf{R}} \in \mathsf{Mod}_{\mathsf{R}}$); and $x.\texttt{BCAS}(e, e, o_{\mathsf{RMW}})$ is an (always successful) blocking CAS. The blocking instructions are implementable using busy loops, but, as observed in [Lahav and Margalit 2019], they make the robustness analysis more expressive as they allow one to mask benign robustness violations: it often happens that stale values can be read in the busy loop, while the use of the blocking instructions instead of the busy loops ensures robustness.

Sequential and concurrent programs are defined as follows.

Definition 2.1. A *sequential program sPr* is a function from a set of the form $\{0, 1, \ldots, N\}$ (the possible values of the program counter) to instructions. We denote by SProg the set of all sequential programs. A *concurrent program Pr* is a mapping from Tid to SProg. For simplicity, we assume that the sets of registers mentioned in the components of concurrent programs are pairwise disjoint.

Concurrent programs (which we also shortly call *programs*) are top-level parallel compositions of sequential programs. In our examples, we often write sequential programs as sequences of instructions delimited by ";" or line breaks, and use '$\|$' for parallel composition.

2.2 From Programs to Labeled Transition Systems

Next, we show how programs are read as labeled transition systems. At this stage, the memory system, responsible of determining which values can be read at each point, is left parametric. To define the alphabet of these LTSs we use *event labels* and *sequential transition labels*:

Definition 2.2. An *event label l* takes one of the following forms: $\mathtt{R}(o_{\mathsf{R}}, x, v_{\mathsf{R}})$ (read label), $\mathtt{W}(o_{\mathsf{W}}, x, v_{\mathsf{W}})$ (write label), $\mathtt{RMW}(o_{\mathsf{RMW}}, x, v_{\mathsf{R}}, v_{\mathsf{W}})$ (RMW label), $\mathtt{R}^{\star}(o_{\mathsf{R}}, x, v_{\mathsf{R}})$ (wait/failed CAS label), $\mathtt{F}(o_{\mathsf{F}})$ (fence label), or $\mathtt{CTRL}$ (control label), where $x \in \mathsf{Loc}$, $v_{\mathsf{R}}, v_{\mathsf{W}} \in \mathsf{Val}$, and $o_{\mathsf{X}} \in \mathsf{Mod}_{\mathsf{X}}$ (for $\mathtt{X} \in \{\mathtt{R}, \mathtt{W}, \mathtt{RMW}, \mathtt{F}\}$). The functions $\mathtt{typ}$, $\mathtt{loc}$, $\mathtt{val}_{\mathsf{R}}$, $\mathtt{val}_{\mathsf{W}}$, and $\mathtt{mod}$ return (when applicable) the type ($\mathtt{R}/\mathtt{W}/\mathtt{RMW}/\mathtt{R}^{\star}/\mathtt{F}/\mathtt{CTRL}$), location ($x$), read value ($v_{\mathsf{R}}$), written value ($v_{\mathsf{W}}$), and access mode ($o_{\mathsf{X}}$) of a given event label. We denote by Lab the set of all event labels.

Definition 2.3. A *sequential transition label L* is a triple of the form $\langle l_{\varepsilon}, R_{\mathsf{in}}, R_{\mathsf{out}} \rangle$, where $l_{\varepsilon} \in \mathsf{Lab} \cup \{\varepsilon\}$ and $R_{\mathsf{in}}, R_{\mathsf{out}} \subseteq \mathsf{Reg}$. The functions $\mathtt{lab}$, $\mathtt{R}_{\mathsf{in}}$, and $\mathtt{R}_{\mathsf{out}}$ return the components of a given sequential transition label. The functions $\mathtt{typ}$, $\mathtt{loc}$, $\mathtt{val}_{\mathsf{R}}$, $\mathtt{val}_{\mathsf{W}}$, and $\mathtt{mod}$ are lifted to sequential transition labels in the obvious way (undefined when $\mathtt{lab}(L) = \varepsilon$). We denote by TLab the set of all sequential transition labels.

Event labels record the memory operation performed at each execution step. Distinguishing normal reads and $\mathtt{R}^{\star}$-reads which arise from wait and (failing) CAS instructions is instrumental in our observational robustness notion (we will not allow "speculative" $\mathtt{R}^{\star}$-reads). In turn, sequential transition labels also include silent program steps (with $\mathtt{lab}(L) = \varepsilon$), and further record the input registers R_{in} (registers that were read in each step) and the output registers R_{out} (registers that were written to in each step). Together with the fact that we have explicit $\mathtt{CTRL}$-labels, this allows us

$$\frac{\begin{array}{c} sPr(pc) = r := e \\ \phi' = \phi[r \mapsto \phi(e)] \\ R_{\mathsf{in}} = \mathsf{Reg}(e) \qquad R_{\mathsf{out}} = \{r\} \end{array}}{\langle pc, \phi \rangle \xrightarrow{\varepsilon, R_{\mathsf{in}}, R_{\mathsf{out}}}_{sPr} \langle pc+1, \phi' \rangle} \qquad \frac{\begin{array}{c} sPr(pc) = \texttt{if } e \texttt{ goto } pc_1, \dots, pc_n \\ \phi(e) \neq 0 \implies pc' \in \{pc_1, \dots, pc_n\} \\ \phi(e) = 0 \implies pc' = pc + 1 \\ l = \mathsf{CTRL} \qquad R_{\mathsf{in}} = \mathsf{Reg}(e) \end{array}}{\langle pc, \phi \rangle \xrightarrow{l, R_{\mathsf{in}}, \emptyset}_{sPr} \langle pc', \phi \rangle} \qquad \frac{\begin{array}{c} sPr(pc) = \texttt{assert}(e) \\ \phi(e) \neq 0 \implies pc' = pc' + 1 \\ \phi(e) = 0 \implies pc' = \bot \\ l = \mathsf{CTRL} \qquad R_{\mathsf{in}} = \mathsf{Reg}(e) \end{array}}{\langle pc, \phi \rangle \xrightarrow{l, R_{\mathsf{in}}, \emptyset}_{sPr} \langle pc', \phi \rangle}$$

$$\frac{\begin{array}{c} sPr(pc) = x.\texttt{store}(e, o_{\mathsf{W}}) \\ l = \mathsf{W}(o_{\mathsf{W}}, x, \phi(e)) \\ R_{\mathsf{in}} = \mathsf{Reg}(e) \end{array}}{\langle pc, \phi \rangle \xrightarrow{l, R_{\mathsf{in}}, \emptyset}_{sPr} \langle pc+1, \phi \rangle} \qquad \frac{\begin{array}{c} sPr(pc) = r := x.\texttt{load}(o_{\mathsf{R}}) \\ l = \mathsf{R}(o_{\mathsf{R}}, x, v) \\ \phi' = \phi[r \mapsto v] \qquad R_{\mathsf{out}} = \{r\} \end{array}}{\langle pc, \phi \rangle \xrightarrow{l, \emptyset, R_{\mathsf{out}}}_{sPr} \langle pc+1, \phi' \rangle}$$

$$\frac{\begin{array}{c} sPr(pc) = r := x.\texttt{FADD}(e, o_{\mathsf{RMW}}) \\ l = \mathsf{RMW}(o_{\mathsf{RMW}}, x, v, v + \phi(e)) \\ \phi' = \phi[r \mapsto v] \\ R_{\mathsf{in}} = \mathsf{Reg}(e) \qquad R_{\mathsf{out}} = \{r\} \end{array}}{\langle pc, \phi \rangle \xrightarrow{l, R_{\mathsf{in}}, R_{\mathsf{out}}}_{sPr} \langle pc+1, \phi' \rangle} \qquad \frac{\begin{array}{c} sPr(pc) = r := x.\texttt{CAS}(e_{\mathsf{R}}, e_{\mathsf{W}}, o_{\mathsf{RMW}}, o_{\mathsf{R}}) \\ l = \mathsf{RMW}(o_{\mathsf{RMW}}, x, \phi(e_{\mathsf{R}}), \phi(e_{\mathsf{W}})) \\ \phi' = \phi[r \mapsto \phi(e_{\mathsf{R}})] \\ R_{\mathsf{in}} = \mathsf{Reg}(e_{\mathsf{R}}) \cup \mathsf{Reg}(e_{\mathsf{W}}) \qquad R_{\mathsf{out}} = \{r\} \end{array}}{\langle pc, \phi \rangle \xrightarrow{l, R_{\mathsf{in}}, R_{\mathsf{out}}}_{sPr} \langle pc+1, \phi' \rangle} \qquad \frac{\begin{array}{c} sPr(pc) = r := x.\texttt{CAS}(e_{\mathsf{R}}, e_{\mathsf{W}}, o_{\mathsf{RMW}}, o_{\mathsf{R}}) \\ l = \mathsf{R}^{\star}(o_{\mathsf{R}}, x, v) \qquad v \neq \phi(e_{\mathsf{R}}) \\ \phi' = \phi[r \mapsto v] \\ R_{\mathsf{in}} = \mathsf{Reg}(e_{\mathsf{R}}) \qquad R_{\mathsf{out}} = \{r\} \end{array}}{\langle pc, \phi \rangle \xrightarrow{l, R_{\mathsf{in}}, R_{\mathsf{out}}}_{sPr} \langle pc+1, \phi' \rangle}$$

$$\frac{\begin{array}{c} sPr(pc) = \texttt{fence}(o_{\mathsf{F}}) \\ l = \mathsf{F}(o_{\mathsf{F}}) \end{array}}{\langle pc, \phi \rangle \xrightarrow{l, \emptyset, \emptyset}_{sPr} \langle pc+1, \phi \rangle} \qquad \frac{\begin{array}{c} sPr(pc) = \texttt{wait}(x = e, o_{\mathsf{R}}) \\ l = \mathsf{R}^{\star}(o_{\mathsf{R}}, x, \phi(e)) \\ R_{\mathsf{in}} = \mathsf{Reg}(e) \end{array}}{\langle pc, \phi \rangle \xrightarrow{l, R_{\mathsf{in}}, \emptyset}_{sPr} \langle pc+1, \phi \rangle} \qquad \frac{\begin{array}{c} sPr(pc) = x.\texttt{BCAS}(e_{\mathsf{R}}, e_{\mathsf{W}}, o_{\mathsf{RMW}}) \\ l = \mathsf{RMW}(o_{\mathsf{RMW}}, x, \phi(e_{\mathsf{R}}), \phi(e_{\mathsf{W}})) \\ R_{\mathsf{in}} = \mathsf{Reg}(e_{\mathsf{R}}) \cup \mathsf{Reg}(e_{\mathsf{W}}) \end{array}}{\langle pc, \phi \rangle \xrightarrow{l, R_{\mathsf{in}}, \emptyset}_{sPr} \langle pc+1, \phi \rangle}$$

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

to observe dependencies between actions in the LTS induced by a program, with no need to refer back to the code itself. Note that for the language constructs introduced above, R_{out} will always be either empty or a singleton.

The next definition formally associates sequential programs with LTSs. In the sequel we identify sequential programs with their induced LTSs (when writing, e.g., $sPr.\mathsf{Q}$ and $\rightarrow_{sPr}$).

Definition 2.4. The LTS (over the alphabet TLab) induced by sPr is defined as follows:

- Its states are pairs $q = \langle pc, \phi \rangle$, where $pc \in \{0, \dots, N\} \cup \{\bot\}$ is the *program counter* (where $N \geq 0$ and $\bot$ denotes an "error state") and $\phi : \mathsf{Reg} \to \mathsf{Val}$ is the *local store* assigning values to registers. We assume that ϕ is extended to apply on expressions e in a standard way that reflects the (unspecified) expression semantics.
- The initial state is $\langle 0, \lambda r.\, 0 \rangle$.
- The transitions are given in Fig. 2 (providing formal meaning to the descriptions above).

Example 2.5. We present the LTS induced by a simple sequential program. Let $\mathsf{Val} = \{0, \dots, 4\}$, $\mathsf{Loc} = \{\mathsf{x}\}$ and $\mathsf{Reg} = \{\mathsf{a}\}$. We evaluate expressions of the form $r = e$ to be 1 if $\phi(r) = \phi(e)$ and 0 otherwise.

```
sPr ≜
  0 : x.store(1, rel)
  1 : a := x.load(acq)
  2 : assert(a = 1)
```

$$\begin{array}{l} sPr.\mathsf{Q} = \{0,1,2,3,\bot\} \times \{[\mathsf{a} \mapsto v] \mid v \in \mathsf{Val}\} \qquad sPr.\mathsf{q}_{\emptyset} = \langle 0, [\mathsf{a} \mapsto 0] \rangle \\ \rightarrow_{sPr} = \{\langle 0, [\mathsf{a} \mapsto v] \rangle \xrightarrow{\mathsf{W(rel,x,1)}, \emptyset, \emptyset}_{sPr} \langle 1, [\mathsf{a} \mapsto v] \rangle \mid v \in \mathsf{Val}\} \cup \\ \qquad \{\langle 1, [\mathsf{a} \mapsto v] \rangle \xrightarrow{\mathsf{R(acq,x,}u\mathsf{)}, \emptyset, \{\mathsf{a}\}}_{sPr} \langle 2, [\mathsf{a} \mapsto u] \rangle \mid v, u \in \mathsf{Val}\} \cup \\ \qquad \{\langle 2, [\mathsf{a} \mapsto 1] \rangle \xrightarrow{\mathsf{CTRL}, \{\mathsf{a}\}, \emptyset}_{sPr} \langle 3, [\mathsf{a} \mapsto 1] \rangle\} \cup \\ \qquad \{\langle 2, [\mathsf{a} \mapsto v] \rangle \xrightarrow{\mathsf{CTRL}, \{\mathsf{a}\}, \emptyset}_{sPr} \langle \bot, [\mathsf{a} \mapsto v] \rangle \mid v \in \mathsf{Val} \setminus \{1\}\} \end{array}$$

The last definition is straightforwardly lifted to the concurrent setting. (Again, we will identify concurrent programs with their induced LTSs.)

Definition 2.6. The LTS induced by a (concurrent) program Pr is the LTS over the alphabet $\mathsf{Tid} \times \mathsf{TLab}$ whose states are tuples $\overline{q} \in \prod_{\tau \in \mathsf{Tid}} Pr(\tau).\mathsf{Q}$; its initial state is $\lambda\tau.\ Pr(\tau).\mathsf{q}_0$; and its transitions are interleaved transitions of Pr's components, formally given by:

$$\frac{\overline{q}(\tau) \xrightarrow{L}_{Pr(\tau)} q'}{\overline{q} \xrightarrow{\tau,L}_{Pr} \overline{q}[\tau \mapsto q']}$$

On the level of programs, the values read by loads and RMWs are completely arbitrary. They are constrained by synchronizing a program with a *memory system*, and obtaining a *concurrent system*, as defined next.

Definition 2.7. A *memory system* $\mathcal{M}$ is an LTS over the alphabet $\mathsf{Tid} \times \mathsf{TLab}$. The concurrent system induced by a program Pr and a memory system $\mathcal{M}$, denoted by $Pr \shortparallel \mathcal{M}$, is the LTS over the alphabet $\mathsf{Tid} \times \mathsf{TLab}$ whose states are pairs in $Pr.\mathsf{Q} \times \mathcal{M}.\mathsf{Q}$; its initial state is $\langle Pr.\mathsf{q}_0, \mathcal{M}.\mathsf{q}_0 \rangle$; and its transitions are synchronized transitions of Pr and $\mathcal{M}$, formally given by:

$$\frac{\overline{q} \xrightarrow{\tau,L}_{Pr} \overline{q}' \qquad q_{\mathcal{M}} \xrightarrow{\tau,L}_{\mathcal{M}} q'_{\mathcal{M}}}{\langle \overline{q}, q_{\mathcal{M}} \rangle \xrightarrow{\tau,L}_{Pr \shortparallel \mathcal{M}} \langle \overline{q}', q'_{\mathcal{M}} \rangle}$$

The most well-known memory system is the one of sequential consistency, denoted here by SC. This memory system simply tracks the most recent value written to each location. Formally, it is defined as follows.

Definition 2.8. The memory system SC is given by: $\mathsf{SC.Q} \triangleq \mathsf{Loc} \to \mathsf{Val}$, $\mathsf{SC.q}_0 \triangleq \lambda x \in \mathsf{Loc}.\ 0$ and $\to_{\mathsf{SC}}$ is given by:

$$\frac{\begin{array}{c}\mathtt{typ}(L) \in \{\mathsf{R}, \mathsf{R}^\star\} \\ M(\mathtt{loc}(L)) = \mathtt{val}_{\mathsf{R}}(L)\end{array}}{M \xrightarrow{\tau,L}_{\mathsf{SC}} M} \qquad \frac{\begin{array}{c}\mathtt{typ}(L) = \mathsf{W} \\ M' = M[\mathtt{loc}(L) \mapsto \mathtt{val}_{\mathsf{W}}(L)]\end{array}}{M \xrightarrow{\tau,L}_{\mathsf{SC}} M'} \qquad \frac{\begin{array}{c}\mathtt{typ}(L) = \mathsf{RMW} \\ M(\mathtt{loc}(L)) = \mathtt{val}_{\mathsf{R}}(L) \\ M' = M[\mathtt{loc}(L) \mapsto \mathtt{val}_{\mathsf{W}}(L)]\end{array}}{M \xrightarrow{\tau,L}_{\mathsf{SC}} M'} \qquad \frac{\mathtt{typ}(L) \notin \{\mathsf{R}, \mathsf{W}, \mathsf{RMW}, \mathsf{R}^\star\}}{M \xrightarrow{\tau,L}_{\mathsf{SC}} M}$$

Finally, we can define what it means for a program to fail under a memory system $\mathcal{M}$.

Definition 2.9. A state $\overline{q} \in Pr.\mathsf{Q}$ is *failing* if $\overline{q}(\tau) = \langle \bot, \phi \rangle$ for some $\tau \in \mathsf{Tid}$ and local store ϕ.

Definition 2.10. A program Pr *may fail under a memory system* $\mathcal{M}$ if $\langle \overline{q}, q_{\mathcal{M}} \rangle$ is reachable in $Pr \shortparallel \mathcal{M}$ for some failing state $\overline{q} \in Pr.\mathsf{Q}$ and $q_{\mathcal{M}} \in \mathcal{M}.\mathsf{Q}$.

Our assumption that the program is finite-data (in particular, we assume that Val is finite) ensures that the question whether a given program may fail under SC is decidable (it is PSPACE-complete [Kozen 1977]). The goal of robustness analysis against a memory system $\mathcal{M}$ (weaker than SC) is to reduce the question of whether a given program Pr may fail under $\mathcal{M}$ to the question of whether Pr may fail under SC.

3 DECLARATIVE CONCURRENCY SEMANTICS USING EXECUTION GRAPHS

In this section we formulate the C11-style memory model, which we call RC20, that we consider in this paper. Like C11 [Batty et al. 2011] and its rectified version RC11 [Lahav et al. 2017], RC20 is defined declaratively—it identifies program behaviors with partially ordered program traces, called *execution graphs*, that meet certain consistency constraints. In §3.1 we formally define execution graphs and present RC20's consistency notion. In §3.2, we formulate the correspondence between programs and sets of execution graphs.

3.1 Execution Graphs and RC20-Consistency

We define execution graphs, starting from their ingredients: an underlying set of *events*, a *labeling function* assigning an event label to each event, a *dependency relation* between events in the same thread, a *reads-from* mapping determining the write event from which each read event reads its value, and a *modification order* totally ordering the writes to each location.

Definition 3.1. An *event* e is a pair $\langle \tau, sn \rangle$, where $\tau \in \mathsf{Tid} \uplus \{\bot\}$ is a thread identifier ($\bot$ is used for initialization events) and $sn \in \mathbb{N}$ is a serial numbers. The functions $\mathtt{tid}$ and $\mathtt{sn}$ return the thread identifier and serial number of an event. We denote the set of all events by Event. Given a set $E \subseteq \mathsf{Event}$ and $\tau \in \mathsf{Tid}$, we write E^{τ} for $\{e \in E \mid \mathtt{tid}(e) = \tau\}$.

Definition 3.2. Let E be a set of events and $lab : E \to \mathsf{Lab}$ be a labeling function (Lab is defined in Def. 2.2).

- A relation dp is a *dependency relation* for E and lab if $dp \subseteq E \times E$ and $\langle e_1, e_2 \rangle \in dp$ implies $\mathtt{tid}(e_1) = \mathtt{tid}(e_2)$, $\mathtt{sn}(e_1) < \mathtt{sn}(e_2)$, and $\mathtt{typ}(lab(e_1)) \in \{\mathtt{R}, \mathtt{RMW}, \mathtt{R}^\star\}$.
- A relation rf is a *reads-from* relation for E and lab if the following hold:
  - If $\langle w, r \rangle \in rf$, then $w, r \in E$, $\mathtt{typ}(lab(w)) \in \{\mathtt{W}, \mathtt{RMW}\}$, $\mathtt{typ}(lab(r)) \in \{\mathtt{R}, \mathtt{RMW}, \mathtt{R}^\star\}$, $\mathtt{loc}(lab(w)) = \mathtt{loc}(lab(r))$, and $\mathtt{val}_{\mathtt{W}}(lab(w)) = \mathtt{val}_{\mathtt{R}}(lab(r))$.
  - If $\langle w_1, r \rangle, \langle w_2, r \rangle \in rf$, then $w_1 = w_2$ (that is, rf^{-1} is functional).
  - For every $r \in \{e \in E \mid \mathtt{typ}(lab(e)) \in \{\mathtt{R}, \mathtt{RMW}, \mathtt{R}^\star\}\}$, we have $\langle w, r \rangle \in rf$ for some $w \in E$.
- A relation mo is a *modification order* for E and lab if it is a disjoint union of relations $\{mo_x\}_{x \in \mathsf{Loc}}$, where each mo_x is a strict total order on

$$\{w \in E \mid \mathtt{typ}(lab(w)) \in \{\mathtt{W}, \mathtt{RMW}\} \land \mathtt{loc}(lab(w)) = x\}.$$

Definition 3.3. An *execution graph* is a tuple $G = \langle E, lab, dp, rf, mo \rangle$ where E is a finite set of events, $lab : E \to \mathsf{Lab}$ is a labeling function, and dp, rf, and mo are respectively a dependency relation, a reads-from relation, and a modification order for E and lab. We denote by EGraph the set of all execution graphs. The components of G are denoted by $G.\mathsf{E}$, $G.\mathtt{lab}$, $G.\mathtt{dp}$, $G.\mathtt{rf}$, and $G.\mathtt{mo}$. We write $G.\mathtt{typ}$ for $\lambda e \in G.\mathsf{E}.\ \mathtt{typ}(G.\mathtt{lab}(e))$ and use similar notations for $\mathtt{loc}$, $\mathtt{val}_{\mathtt{R}}$, $\mathtt{val}_{\mathtt{W}}$, and $\mathtt{mod}$. We use the following notations for restricted sets of events:

$$G.\mathsf{R} \triangleq \{e \in G.\mathsf{E} \mid G.\mathtt{typ}(e) \in \{\mathtt{R}, \mathtt{RMW}, \mathtt{R}^\star\}\} \qquad G.\mathsf{W} \triangleq \{e \in G.\mathsf{E} \mid G.\mathtt{typ}(e) \in \{\mathtt{W}, \mathtt{RMW}\}\}$$

$$\text{For } \alpha \in \{\mathtt{RMW}, \mathtt{F}, \mathtt{R}^\star, \mathtt{CTRL}\} : \quad G.\alpha \triangleq \{e \in G.\mathsf{E} \mid G.\mathtt{typ}(e) = \alpha\}$$

We employ subscripts and superscripts to restrict sets of events to certain properties (e.g., $G.\mathsf{W}_x^{\sqsupseteq \mathtt{rel}} = \{w \in G.\mathsf{W} \mid G.\mathtt{loc}(w) = x \land G.\mathtt{mod}(w) \sqsupseteq \mathtt{rel}\}$).

We assume that execution graphs are always initialized, as defined next.

Definition 3.4. The set of Init initialization events consists of an event w_x^{Init} for every $x \in \mathsf{Loc}$ with $\mathtt{tid}(w_x^{\mathsf{Init}}) = \bot$ and $\mathtt{sn}(w_x^{\mathsf{Init}}) = 0$. The initial labeling function lab^{Init} is the function assigning the label $\mathtt{W}(\mathtt{rlx}, x, 0)$ to each every initialization event $w_x^{\mathsf{Init}} \in \mathsf{Init}$. The initial execution graph $G_\emptyset$ is given by $G_\emptyset \triangleq \langle \mathsf{Init}, lab^{\mathsf{Init}}, \emptyset, \emptyset, \emptyset \rangle$. An execution graph G is *initialized* if it extends $G_\emptyset$ (that is: $\mathsf{Init} \subseteq G.\mathsf{E}$ and $lab^{\mathsf{Init}} \subseteq G.\mathtt{lab}$).

Our representation of events induces a *program order*, in which events of the same thread are ordered according to their serial numbers and initialization events precede all other events (i.e., $\langle \tau_1, sn_1 \rangle < \langle \tau_2, sn_2 \rangle$ if $\tau_1 = \tau_2 \neq \bot$ and $sn_1 < sn_2$, or $\tau_1 = \bot$ and $\tau_2 \neq \bot$). We denote by $G.\mathsf{po}$ the restriction of this order to $G.\mathsf{E}$ (i.e., $G.\mathsf{po} \triangleq \{\langle e_1, e_2 \rangle \in G.\mathsf{E} \times G.\mathsf{E} \mid e_1 < e_2\}$).

To define RC20-consistent execution graphs, we need several additional derived relations. First, we define the *synchronizes with* relation and the *happens-before* relation:

$$G.\mathtt{sw} \triangleq [G.\mathsf{E}^{\sqsupseteq \mathtt{rel}}] \mathbin{;} ([G.\mathsf{F}] \mathbin{;} G.\mathsf{po})^{?} \mathbin{;} G.\mathtt{rf}^{+} \mathbin{;} (G.\mathsf{po} \mathbin{;} [G.\mathsf{F}])^{?} \mathbin{;} [G.\mathsf{E}^{\sqsupseteq \mathtt{acq}}] \qquad (\textit{synchronizes with})$$

$$G.\mathtt{hb} \triangleq (G.\mathsf{po} \cup G.\mathtt{sw})^{+} \qquad (\textit{happens-before})$$

Happens-before consists of all paths formed of program order edges (po) and synchronization edges (sw). The latter contains rf-edges between release writes and acquire reads. Synchronization is also induced by:

(1) rf-edges from relaxed writes that follow release fences;
(2) rf-edges to relaxed reads that are followed by acquire fences; and
(3) *release sequences* (using C11's terminology)—sequences of rf-edges, where all internal events in the sequence are RMWs (necessarily, as they both read and write).

We note that (R)C11's synchronization also includes $[G.\mathsf{W}^{\sqsupseteq \mathtt{rel}}] \mathbin{;} (G.\mathsf{po} \cap G.\mathtt{mo}) \mathbin{;} G.\mathtt{rf}^{+} \mathbin{;} (G.\mathsf{po} \mathbin{;} [G.\mathsf{F}])^{?} \mathbin{;} [G.\mathsf{E}^{\sqsupseteq \mathtt{acq}}]$. To simplify the model, observing that this extension is rarely used, it was recently discarded from the C/C++ memory model [Boehm et al. 2018]. Accordingly, we use here the simpler version of sw.

Second, we use the standard *from-read* relation (a.k.a. *reads-before*), which relates reads with subsequent writes to the same location (write events that are mo-later than the write event that the read event reads its value from):

$$G.\mathtt{fr} \triangleq (G.\mathtt{rf}^{-1} \mathbin{;} G.\mathtt{mo}) \setminus [G.\mathsf{E}] \qquad (\textit{from-read})$$

Now, following (R)C11, RC20-consistency is defined as the conjunction of four constraints:

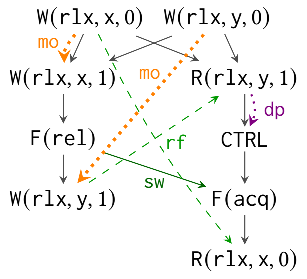
Definition 3.5. An execution graph G is RC20-*consistent* if the following hold:

- $G.\mathtt{mo} \mathbin{;} G.\mathtt{rf}^{?} \mathbin{;} G.\mathtt{hb}^{?}$ is irreflexive. (WRITE COHERENCE)
- $G.\mathtt{fr} \mathbin{;} G.\mathtt{rf}^{?} \mathbin{;} G.\mathtt{hb}$ is irreflexive. (READ COHERENCE)
- $G.\mathtt{fr} \mathbin{;} G.\mathtt{mo}$ is irreflexive. (ATOMICITY)
- $G.\mathsf{po} \cup G.\mathtt{rf}$ is acyclic. (PO-RF)

The first two constraints ensure that the per-location order $(\mathtt{mo} \cup \mathtt{fr}) \mathbin{;} \mathtt{rf}^{?}$ agrees with hb. The third constraint guarantees the atomicity of RMWs (in particular, it implies that two RMWs cannot read from the same event). The last constraint is a conservative strengthening of the original C11 model employed in the RC11 model [Lahav et al. 2017] (following [Boehm and Demsky 2014]) for avoiding the out-of-thin-air problem. To the best of our knowledge, this acyclicity condition is assumed in all C11 verification methods and tools e.g., [Doherty et al. 2019; Kokologiannakis et al. 2017; Vafeiadis and Narayan 2013].

Example 3.6. Consider the following execution graph of the standard message passing test (MP):

```
                 || START :
                 || a := y.load(rlx)
x.store(1, rlx)  || if a = 0 goto START
fence(rel)       || fence(acq)
y.store(1, rlx)  || b := x.load(rlx) // 0
                 || assert(b ≠ 0)
```



Due to the rf-edge from W(rlx, y, 1) to R(rlx, y, 1) we obtain synchronization (sw) from F(rel) to

$\mathtt{F(acq)}$, and thus $\mathtt{hb}$ from $\mathtt{W(rlx, x, 1)}$ to $\mathtt{R(rlx, x, 0)}$. Since $\mathtt{fr}$ goes in the opposite direction, this execution violates READ COHERENCE, and so it is not RC20-consistent. Omitting one (or both) of the fences will remove the synchronization, making this graph RC20-consistent.

There are several differences between RC20 and RC11:

- For simplicity, RC20 does not include SC accesses, whose semantics was rectified in the RC11 model [Lahav et al. 2017]. We are aware of only a few (practical) algorithms that actually employ SC accesses and become wrong when release/acquire accesses are used instead. In these cases, SC-fences can be used instead of SC accesses, although this may incur a certain cost when targeting ARMv8 that has special support for SC accesses.
- C11's SC-fences, whose semantics was rectified in the RC11 model [Lahav et al. 2017], are not included in RC20 as primitives. Instead, in RC20 SC-fences are syntactic sugar for sequences of instructions: $\mathtt{fence(acq)}; \_ := \mathtt{f.FADD(0, acqrel)}; \mathtt{fence(rel)}$ where $\mathtt{f}$ is a distinguished otherwise-unused location. The two fences in the sequence ensure that SC-fences will behave both as an acquire fence and as a release fence for inducing synchronization using the $\mathtt{sw}$ relation (w.r.t. later writes and prior reads). The RMW instruction (FADD) ensures that $\mathtt{hb}$ will totally order the SC-fences in the execution graph, which together with WRITE COHERENCE and READ COHERENCE, provides us with the required guarantees.
- For the simplicity of the presentation, our formulation so far did not discuss C11's non-atomic accesses that have "catch-fire" semantics. These can be easily added (see §6).

Remark 1. Interestingly, the above encoding of SC fences results in a semantics that is strictly stronger (i.e., allowing less behaviors) than RC11's. Our model can be shown to be equivalent to a model with SC fences as primitives that includes $[\mathtt{F^{sc}}]; (\mathtt{po} \cup \mathtt{rf})^+; [\mathtt{F^{sc}}]$ and $[\mathtt{F^{sc}}]; (\mathtt{po} \cup \mathtt{sw})^+; (\mathtt{rf} \cup \mathtt{mo}; \mathtt{rf}^? \cup \mathtt{fr}; \mathtt{rf}^?); (\mathtt{po} \cup \mathtt{sw})^+; [\mathtt{F^{sc}}]$ as components of $\mathtt{hb}$ (together with $\mathtt{po}$ and $\mathtt{sw}$). In turn, RC11 only requires that $[\mathtt{F^{sc}}]; (\mathtt{hb} \cup \mathtt{hb}; (\mathtt{rf} \cup \mathtt{mo}; \mathtt{rf}^? \cup \mathtt{fr}; \mathtt{rf}^?); \mathtt{hb}); [\mathtt{F^{sc}}]$ is acyclic. We conjecture that the existing mapping of RC11 to hardware, as well as the compiler transformations that are sound under RC11, can be validated for our semantics. This will provide: (*i*) stronger guarantees to the programmers with no additional cost; and (*ii*) a more principled and parsimonious approach for the semantics of SC fences (they are precisely captured by $\mathtt{fence(acq)}; \_ := \mathtt{f.FADD(0, acqrel)}; \mathtt{fence(rel)}$) instead of their rather ad-hoc semantics in RC11. The latter is in line with other models (e.g., TSO and the fragment of RC11 consisting of release/acquire atomics and non-atomics) in which an RMW to a distinguished otherwise-unused location is *equivalent* to an SC-fence. We note that previous work on verification under RC11 assumed a similar stronger simplified semantics for SC fences [Dang et al. 2019].

We end this section by recalling the definition of sequential consistency in the declarative framework. This formulation is used in our robustness notions (in §4) that are based on comparing the set of RC20-consistent execution graphs of a given program to the set of SC-consistent execution graphs of that program. We follow the standard definition (see, e.g., [Alglave et al. 2014]) using the derived relation:

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

Definition 3.7. An execution graph G is SC-*consistent* if $G.\mathtt{hb}_{\mathsf{SC}}$ is irreflexive.

It is easy to see that SC-consistency implies RC20-consistency.

3.2 From Concurrent Programs to Execution Graphs

In this section we present the definitions needed for formally relating execution graphs with concurrent programs, and use them to define what it means that a given program may fail under

RC20. For example, we will have that the execution graph in Ex. 3.6 is indeed generated by the program in that example and that this program may not fail under RC20. To so do, we define when an execution graph G is *generated by a program Pr with a final state $\overline{q}$* by using a special memory system, called FG (for *free graphs*). This system sets no restrictions whatsoever in its transitions (every sequence of pairs in $\mathsf{Tid} \times \mathsf{TLab}$ is a trace of FG), but rather maintains in its states the current execution graph. Then, the execution graphs generated by a program Pr with final state $\overline{q}$ are defined as the execution graphs reachable by FG when FG executes a trace of Pr that reaches $\overline{q}$.

To define FG, we use the following notation for adding a fresh event to an execution graph. It "receives as input" the current execution graph G, the thread identifier τ of the new event, the label l of the new event, the set D of events on which the new event depends, and the write event w that the new event reads from if it is a read, or placed as the $\mathtt{mo}$-successor of if it is a write. When the new event is an RMW ($\mathtt{typ}(l) = \mathtt{RMW}$), its $\mathtt{rf}$-source and $\mathtt{mo}$-predecessor must coincide (this is in line with WRITE COHERENCE and ATOMICITY). When the new event is a fence or a control event, we do not need the event w.

Notation 3.8. *For $G \in \mathsf{EGraph}$ and $\tau \in \mathsf{Tid}$, $\mathsf{next}(G, \tau)$ denotes the "next available event" for thread τ in G, given by $\mathsf{next}(G,\tau) \triangleq \langle \tau, \max\{\mathsf{sn}(e) \mid e \in G.\mathtt{E}^{\tau}\} + 1\rangle$. For $G \in \mathsf{EGraph}$, $\tau \in \mathsf{Tid}$, $l \in \mathsf{Lab}$, $D \subseteq \mathsf{Event}$, and $w \in \mathsf{Event} \cup \{\bot\}$, $\mathsf{add}(G, \tau, l, D, w)$ denotes the tuple $\langle E', lab', dp', rf', mo'\rangle$ defined as follows, where $e = \mathsf{next}(G, \tau)$:*

$$E' = G.\mathtt{E} \cup \{e\} \qquad rf' = \begin{cases} G.\mathtt{rf} \cup \{\langle w, e\rangle\} & \mathtt{typ}(l) \in \{\mathtt{R}, \mathtt{RMW}, \mathtt{R}^{\star}\} \\ G.\mathtt{rf} & \textit{otherwise} \end{cases}$$

$$lab' = G.\mathtt{lab} \cup \{e \mapsto l\}$$

$$dp' = G.\mathtt{dp} \cup (D \times \{e\})$$

$$mo' = \begin{cases} G.\mathtt{mo} \cup \{\langle w', e\rangle \mid \langle w', w\rangle \in G.\mathtt{mo}^{?}\} \cup \{\langle e, w'\rangle \mid \langle w, w'\rangle \in G.\mathtt{mo}\} & \mathtt{typ}(l) \in \{\mathtt{W}, \mathtt{RMW}\} \\ G.\mathtt{mo} & \textit{otherwise} \end{cases}$$

To properly set the dependency relation between the events of the graph, together with the current execution graph, FG carries a *dependency store* in its state, which records for every register r the graph events that correspond to the reads on which the value of r depends.

Definition 3.9. A *dependency store* is a function $\psi : \mathsf{Reg} \rightarrow \mathcal{P}(\mathsf{Event})$. For $R \subseteq \mathsf{Reg}$, we write $\psi(R)$ for the set $\bigcup_{r \in R} \psi(r)$, and $\psi[R \mapsto E]$ for the dependency store ψ' that is identical to ψ (possibly) except for $\psi'(r) = E$ for every $r \in R$.

Having $e \in \psi(r)$ means that the value of r depends on the event e. For example, after executing $\mathtt{a} := \mathtt{x.load(rlx)}$ we will have $\psi(\mathtt{a}) = \{e\}$, where e is the (fresh) read event generated for this load instruction. For $\mathtt{a} := \mathtt{x.load(rlx)}; \mathtt{b} := \mathtt{x.load(rlx)}; \mathtt{c} := \mathtt{a} + \mathtt{b}$, we will have *two* events in $\psi(\mathtt{c})$. This allows us to generate appropriate dependency edges, when, say, $\mathtt{y.store(c, rlx)}$ comes next. (Our assumption that different threads use disjoint registers simplifies the definition of ψ.)

Now, we define the memory system FG.

Definition 3.10. The memory system FG is given by:

- Its states are pairs $\langle G, \psi\rangle$, where G is an execution graph and ψ is a dependency store.
- Its initial state is $\langle G_{\emptyset}, \psi_{\emptyset}\rangle$, where $G_{\emptyset}$ is the initial execution graph (as defined in Def. 3.4) and $\psi_{\emptyset} \triangleq \lambda r.\ \emptyset$ is the initial dependency store.
- Its transitions are given by:

$$\frac{\begin{array}{c}\mathtt{typ}(L) \in \{\mathsf{R}, \mathsf{W}, \mathsf{RMW}, \mathsf{R}^\star\} \\ w \in G.\mathsf{W}_{\mathtt{loc}(L)} \\ \mathtt{typ}(L) \in \{\mathsf{R}, \mathsf{RMW}, \mathsf{R}^\star\} \implies G.\mathtt{val}_\mathsf{W}(w) = \mathtt{val}_\mathsf{R}(L) \\ G' = \mathtt{add}(G, \tau, \mathtt{lab}(L), \psi(\mathsf{R}_{\mathsf{in}}(L)), w) \\ \psi' = \psi[\mathsf{R}_{\mathsf{out}}(L) \mapsto \{\mathsf{next}(G, \tau)\}]\end{array}}{\langle G, \psi\rangle \xrightarrow{\tau, L}_{\mathsf{FG}} \langle G', \psi'\rangle}$$

$$\frac{\begin{array}{c}\mathtt{typ}(L) \in \{\mathsf{F}, \mathsf{CTRL}\} \\ G' = \mathtt{add}(G, \tau, \mathtt{lab}(L), \psi(\mathsf{R}_{\mathsf{in}}(L)), \bot)\end{array}}{\langle G, \psi\rangle \xrightarrow{\tau, L}_{\mathsf{FG}} \langle G', \psi\rangle}$$

$$\frac{\begin{array}{c}\mathtt{lab}(L) = \varepsilon \\ \psi' = \psi[\mathsf{R}_{\mathsf{out}}(L) \mapsto \psi(\mathsf{R}_{\mathsf{in}}(L))]\end{array}}{\langle G, \psi\rangle \xrightarrow{\tau, L}_{\mathsf{FG}} \langle G, \psi'\rangle}$$

We use FG to formally associate execution graphs with programs.

Definition 3.11. Let Pr be a program. An execution graph G is:

- *generated by Pr with final state $\overline{q}$ and dependency store ψ* if $\langle \overline{q}, G, \psi\rangle$ is reachable in $Pr \parallel \mathsf{FG}$.
- *generated by Pr with final state $\overline{q}$* if $\langle \overline{q}, G, \psi\rangle$ is reachable in $Pr \parallel \mathsf{FG}$ for *some* ψ.
- *generated by Pr* if $\langle \overline{q}, G, \psi\rangle$ is reachable in $Pr \parallel \mathsf{FG}$ for *some* $\overline{q}$ and ψ.

Using the last definition, the semantics to a program Pr under the RC20 model is taken to be the set of RC20-consistent execution graphs that are generated by Pr. For safety under RC20, we use the following definition, which captures when an assertion violation might arise under RC20.

Definition 3.12. A program Pr *may fail under* RC20 if some RC20-consistent execution graph G is generated by Pr with a failing final state $\overline{q}$ (see Def. 2.9).

In addition, we can now state the (well known) equivalence of the SC memory system (Def. 2.8) and its declarative formulation (Def. 3.7) in terms of program safety.

Proposition 3.13. *A program Pr may fail under* SC *(Def. 2.10) iff some* SC*-consistent execution graph G is generated by Pr with a failing final state $\overline{q}$.*

Finally, we will use certain receptiveness properties ensuring that the set of generated execution graphs for a given program is closed under changes of the values read in read events with no outgoing dependency edges. Intuitively, if nothing depends on the difference between two graphs, then this difference cannot be observed by the program.

Definition 3.14. An event r is a *G-irrelevant read* if $G.\mathtt{typ}(e) = \mathsf{R}$ and $r \notin dom(G.\mathsf{dp})$. We denote by $G.\mathsf{IR}$ the set of all G-irrelevant read events.

Definition 3.15. Two execution graphs G and G' are *equal up to a set T of events*, denoted $G \sim_T G'$, if the following hold:

- $G.\mathsf{E} = G'.\mathsf{E}$, $G.\mathsf{dp} = G'.\mathsf{dp}$, and $G.\mathsf{mo} = G'.\mathsf{mo}$.
- $G.\mathtt{typ} = G'.\mathtt{typ}$, $G.\mathtt{loc} = G'.\mathtt{loc}$, $G.\mathtt{val}_\mathsf{W} = G'.\mathtt{val}_\mathsf{W}$, and $G.\mathtt{mod} = G'.\mathtt{mod}$.
- $G.\mathtt{val}_\mathsf{R}(r) = G'.\mathtt{val}_\mathsf{R}(r)$ for every $r \notin T$.
- $G.\mathsf{rf} \,;\, [\mathsf{Event} \setminus T] = G'.\mathsf{rf} \,;\, [\mathsf{Event} \setminus T]$.

We say that G and G' are *observationally equivalent* if $G \sim_{G.\mathsf{IR}} G'$.

The following property ensures that the set of execution graphs generated by a program is closed under observational equivalence.

Definition 3.16. Two program states $\overline{q}, \overline{q}' \in Pr.\mathsf{Q}$ are *equivalent up to a set $R \subseteq \mathsf{Reg}$*, denoted by $\overline{q} \sim_R \overline{q}'$, if $\overline{q}(\tau) = \langle pc, \phi\rangle$ implies that $\overline{q}'(\tau) = \langle pc, \phi'\rangle$ for some local store ϕ' satisfying $\phi'(r) = \phi(r)$ for every $r \notin R$.

Lemma 3.17. *Suppose that $G \sim_T G'$ for some $T \subseteq G.\mathsf{IR}$. If $\langle \overline{q}, G, \psi\rangle$ is reachable in $Pr \parallel \mathsf{FG}$, then $\langle \overline{q}', G', \psi\rangle$ is reachable in $Pr \parallel \mathsf{FG}$ for some program state $\overline{q}' \in Pr.\mathsf{Q}$ such that $\overline{q} \sim_R \overline{q}'$ for $R = \{r \in \mathsf{Reg} \mid \psi(r) \cap T \neq \emptyset\}$.*

4 ROBUSTNESS AGAINST RC20

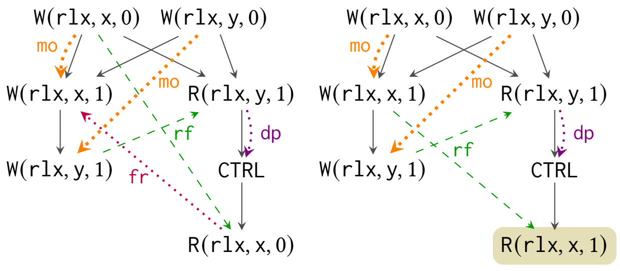
In this section we introduce the two robustness notions studied in this paper: robustness against RC20 and observational robustness against RC20. We show how they can be reduced to certain properties of SC-*consistent* execution graphs that are generated by the given program. Since we only consider robustness against RC20, we will refer to these properties simply as *robustness* and *observational robustness*.

Robustness requires that RC20-consistency and SC-consistency coincide for execution graphs generated by the program:

Definition 4.1. A program Pr is *robust* if all RC20-consistent execution graphs generated by Pr are also SC-consistent.

Example 4.2. It is easy to see that the MP in Ex. 3.6 is robust. The same program without fences, however, is not robust as the RC20-consistent but SC-inconsistent execution graph on the middle demonstrates (the graph on the right is discussed in Ex. 4.15):

```
                   ||  START :
                   ||  a := y.load(rlx)
x.store(1, rlx)    ||  if a = 0 goto START
y.store(1, rlx)    ||  b := x.load(rlx)
                   ||  assert(b ≠ 0)
```



The above robustness notion (w.r.t. different weak memory models) is also known as *trace-robustness* [Bouajjani et al. 2013] or *execution-graph-robustness* [Lahav and Margalit 2019]. It implies *state-robustness*: if some RC20-consistent execution graph G is generated with a final program state $\overline{q}$, then $\langle \overline{q}, M\rangle$ is reachable in $Pr \shortparallel$ SC for some $M : \mathsf{Loc} \to \mathsf{Val}$. In particular, the following property ensures that safety verification of robust programs may assume SC semantics.

Proposition 4.3. *A robust program may fail under* RC20 *(Def. 3.12) iff it may fail under* SC *(Def. 2.10).*

Our first main theorem formulates a technical property of execution graphs that holds for all SC-consistent graphs of a given program *iff* that program is robust. It allows us to avoid the universal quantification over RC20-consistent execution graphs of Def. 4.1 and paves the way for deciding robustness via reachability analysis under SC. (The next section clarifies this reduction.)

Notation 4.4. *$G.w_x^{\max}$ denotes the $G.\mathtt{mo}$-maximal write to x in G (i.e., $G.w_x^{\max} \triangleq \max_{G.\mathtt{mo}} G.\mathsf{W}_x$).*

Definition 4.5. A *non-robustness witness* for a program Pr is a tuple $\langle \overline{q}, G, \tau, L\rangle$ such that the following hold:

- G is generated by Pr with final state $\overline{q}$.
- G is SC-consistent.
- $\overline{q}$ enables the transition $\langle \tau, L\rangle$.
- $\mathsf{typ}(L) \in \{\mathsf{R}, \mathsf{W}, \mathsf{RMW}, \mathsf{R}^\star\}$.
- $G.w_{\mathtt{loc}(L)}^{\max} \in dom(G.\mathtt{hb}_{\mathsf{SC}}^? \,;\, [\mathsf{E}^\tau])$.
- The following hold for some $w \in G.\mathsf{W}_{\mathtt{loc}(L)}$:
  - $w \neq G.w_{\mathtt{loc}(L)}^{\max}$.
  - $w \notin dom(G.\mathtt{mo} \,;\, G.\mathtt{rf}^? \,;\, G.\mathtt{hb}^? \,;\, [\mathsf{E}^\tau])$.
  - If $\mathsf{typ}(L) \in \{\mathsf{W}, \mathsf{RMW}\}$, then $w \notin dom(G.\mathtt{rf} \,;\, [G.\mathsf{RMW}])$.
  - If $\mathsf{typ}(L) \in \{\mathsf{RMW}, \mathsf{R}^\star\}$, then $G.\mathtt{val}_{\mathsf{W}}(w) = \mathtt{val}_{\mathsf{R}}(L)$.

Theorem 4.6. *A program Pr is not robust iff there exists a non-robustness witness for Pr.*

A non-robustness witness forms a “borderline execution” of a given program—an execution graph G of the program that is still SC-consistent, but one of the possible program steps after generating G results in an RC20-consistent but SC-inconsistent graph. Roughly speaking, this scenario happens when (under SC) some read/write/RMW step (labeled with L) accessing location x by thread τ *must* be executed after the maximal write to x in G (namely, there is a $G.\mathsf{hb}_{\mathsf{SC}}$-path from $G.w_x^{\mathsf{max}}$ to some event of thread τ), but some other write w to x in G has no $G.\mathsf{mo}\,;G.\mathtt{rf}^?\,;G.\mathsf{hb}^?$-path to thread τ. The latter condition ensures that thread τ is not “aware” of a write that is later than w, implying that under RC20 we may extend G to the execution graph $G' = \mathtt{add}(G, \tau, \mathtt{lab}(L), \_, w)$. Since $G.w^{\mathsf{max}}_{\mathtt{loc}(L)} \in \mathit{dom}(G.\mathsf{hb}^?_{\mathsf{SC}}\,;[\mathsf{E}^\tau])$, we will have that G' is SC-inconsistent. More care has to be taken because of the ATOMICITY-condition: if the enabled transition is a write (or an RMW) then RC20 forbids placing the new generated event $\mathtt{next}(G, \tau)$ as the mo-successor of writes that are already read by RMWs, and thus we require $w \notin \mathit{dom}(G.\mathtt{rf}\,;[G.\mathsf{RMW}])$ in this case. Furthermore, for G' to be an execution of the program, we also require that $G.\mathtt{val}_{\mathtt{W}}(w) = \mathtt{val}_{\mathtt{R}}(L)$ in case that $\mathtt{typ}(L) \in \{\mathtt{RMW}, \mathtt{R}^\star\}$. The latter condition ensures that the program indeed allows a step leading from G to G'.

Example 4.7. A non-robustness witness for the (non-fenced) MP program in Ex. 4.2 is given by $\langle \overline{q}, G, \tau, L\rangle$, where $\overline{q}$ is the state of the program after the first thread is fully executed and the second thread is just before the load of $\mathtt{x}$ (with $\mathtt{a} = 1$); G is the execution graph obtained from the one presented in the example above by removing the last read event of the second thread; τ is the identifier of the second thread; and L is any transition label with $\mathtt{typ}(L) = \mathtt{R}$, $\mathtt{loc}(L) = \mathtt{x}$, $\mathtt{mod}(L) = \mathtt{rlx}$, $\mathsf{R}_{\mathsf{in}}(L) = \emptyset$, and $\mathsf{R}_{\mathsf{out}}(L) = \{\mathtt{b}\}$. In particular, we have $G.w_{\mathtt{x}}^{\mathsf{max}} \in \mathit{dom}(G.\mathsf{hb}^?_{\mathsf{SC}}\,;[\mathsf{E}^\tau])$ ($G.w_{\mathtt{x}}^{\mathsf{max}}$ is the event labeled with $\mathtt{W}(\mathtt{rlx}, \mathtt{x}, 1)$), and for w being the initial write event to x, we have $w \notin \mathit{dom}(G.\mathsf{mo}\,;G.\mathtt{rf}^?\,;G.\mathsf{hb}^?\,;[\mathsf{E}^\tau])$.

We note that the standard DRF-theorem ensuring that there are not any weak behaviors for programs that are race free under SC (e.g., by using locks implemented with BCAS instructions) is a simple corollary of the above precise characterization. A non-robustness witness necessarily implies a race under SC between $G.w^{\mathsf{max}}_{\mathtt{loc}(L)}$ and an event that can be added to G in the next step of thread τ.

4.1 Observational Robustness

The above robustness notion is too strict for supporting an important use case of relaxed accesses in which the stale values observed by relaxed accesses (and cannot be observed under SC) are in *speculative reads* that are never used by the program. This use case was discussed in [Sinclair et al. 2017] and demonstrated with the *sequence lock* implementation (seqlock for short), a special locking mechanism used in Linux that avoids writer starvation [Boehm 2012]. The next example shows the challenge seqlock poses to robustness verification.

Example 4.8. Consider the litmus test in Fig. 3, extracted from an efficient implementation of seqlock with a single writer and a single reader. The writer executes two iterations in which it increases the version number ($\mathtt{z}$) to be odd, writes the protected data ($\mathtt{x}$ and $\mathtt{y}$), and publishes the data by increasing the version number to be even again. The reader waits in a loop until it reads an even version number, then (speculatively) reads the data, and reads again the version number to check that it has not changed. If the latter read yields a different version number, the reader never uses the data that was read and retries. Finally, if the reader reads the same version number, it can safely use the data (modeled here as $\mathtt{assert}(\mathtt{a} = \mathtt{b})$). Access modes and fences were carefully picked to ensure correctness while optimizing performance [Boehm 2012]. In particular, note that the read of the protected data is relaxed.

```
z.store(1, rlx)  ‖ START :
fence(rel)       ‖ c1 := z.load(rlx)
x.store(1, rlx)  ‖ if isOdd(c1) goto START
y.store(1, rlx)  ‖ fence(acq)
z.store(2, rel)  ‖ a := x.load(rlx)
                 ‖ b := y.load(rlx)
z.store(3, rlx)  ‖ fence(acq)
fence(rel)       ‖ c2 := z.load(rlx)
x.store(2, rlx)  ‖ if c1 ≠ c2 goto START
y.store(2, rlx)  ‖ assert(a = b)
z.store(4, rel)  ‖
```

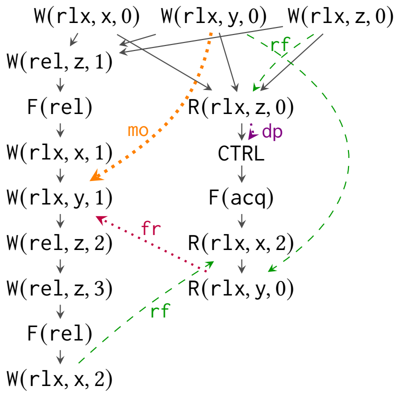


Fig. 3. Seqlock litmus test.

A robustness violation—an execution graph of the program that is RC20-consistent but SC-inconsistent—is presented on the right in Fig. 3 (to avoid clutter we omit some some mo and fr edges). It is caused by the reader reading an updated value from x but an old (the initial) value from y. After reading an updated value from x, the next acquire fence that the reader will perform (before using the values) will synchronize with the last release fence of the writer (creating an sw-edge), so that reading z = 0 again is forbidden (the reader can only read z ≥ 3), and the (taint) data will be necessarily discarded. We note that standard *state* robustness does not hold here as well: a program state in which the reader has a = 2 and b = 0 is reachable under RC20 but not under SC.

To support this use case, we define a weaker robustness notion, which we refer to as *observational robustness*. This notion allows non-SC execution graphs of the program, provided that nothing depends on events that read stale values. To formally define this notion, we adapt the declarative formulation of SC to ignore fr-edges induced by read events with no outgoing dependencies. For that we use two additional derived relations:

$$G.\mathtt{fr}^{\mathsf{dp}} \triangleq [dom(G.\mathsf{dp}) \cup G.\mathsf{R}^{\star}] \mathbin{;} G.\mathtt{fr} \qquad\qquad G.\mathsf{hb}^{\mathsf{dp}}_{\mathsf{SC}} \triangleq (G.\mathsf{po} \cup G.\mathtt{rf} \cup G.\mathtt{mo} \cup G.\mathtt{fr}^{\mathsf{dp}})^{+}$$

While in $\mathsf{hb}_{\mathsf{SC}}$, via $\mathtt{fr}$, a read r is always placed before every write w' written $\mathtt{mo}$-after the write w that r reads from, the $\mathsf{hb}^{\mathsf{dp}}_{\mathsf{SC}}$ relation ignores such orders when (i) $r \notin G.\mathsf{R}^{\star}$ (that is, r does not arise from a wait instruction or a failed CAS) and (ii) no event depends on r. We note that, ATOMICITY ensures that $[G.\mathsf{RMW}] \mathbin{;} G.\mathtt{fr} \subseteq G.\mathtt{mo}$, so $G.\mathsf{hb}^{\mathsf{dp}}_{\mathsf{SC}}$ always contains $\mathtt{fr}$-edges from RMW events.

Using $\mathsf{hb}^{\mathsf{dp}}_{\mathsf{SC}}$, observational robustness is naturally defined as follows.

Definition 4.9. A program *Pr* is *observationally robust* if $G.\mathsf{hb}^{\mathsf{dp}}_{\mathsf{SC}}$ is irreflexive in every RC20-consistent execution graph G that is generated by *Pr*.

Example 4.10. Revisiting the examples above, note that the seqlock test (Ex. 4.8) is *observationally* robust. In particular, the execution graph in Fig. 3 has no $\mathsf{hb}^{\mathsf{dp}}_{\mathsf{SC}}$-cycle. A dependency edge from a $\mathsf{R}(\mathtt{rlx}, \mathtt{y}, v)$-event will be only added if the previous read of x is reading the same value v. The (non-fenced) MP program (Ex. 4.7) is not observationally robust. Without the final assert(b ≠ 0) instruction, that program is observationally robust (but not robust).

To see that observational robustness is not overly weak, we establish the analogue of Prop. 4.3, ensuring that verification of observationally robust programs may safely assume SC-semantics.

This easily follows from the fact that the SC-inconsistent execution graphs allowed by observational robustness are always observationally equivalent to SC-consistent ones (as defined in Def. 3.15), which we prove next.

Lemma 4.11. *Let G be an* RC20*-consistent execution graph with $G.\mathsf{hb}^{\mathsf{dp}}_{\mathsf{SC}}$ being irreflexive. Then, G is observationally equivalent to some* SC*-consistent execution graph G'.*

Lemma 4.11 and the receptiveness property (Lemma 3.17) easily entail the analogue of Prop. 4.3.

Proposition 4.12. *An observationally robust program may fail under* RC20 *(Def. 3.12) iff it may fail under* SC *(Def. 2.10).*

Remark 2. Using Lemma 4.11, we have that observational robustness of a program *Pr* implies that every RC20-consistent execution graph generated by *Pr* is observationally equivalent to some SC-consistent execution graph. We note that the converse does not hold. For example, consider the following program:

```
                  || a := y.load(acq) //1 || z.store(1, rel)
x.store(1, rel)   || b := z.load(acq) //0 || c := x.load(acq) //0
y.store(1, rel)   || assert(b = b)        || assert(c = c)
```

The execution graph G that corresponds to the annotated outcome has a cycle in $G.\mathsf{hb}^{\mathsf{dp}}_{\mathsf{SC}}$ (using two $G.\mathtt{fr}^{\mathsf{dp}}$-edges: from the read of z and from the read of x). Nevertheless, by changing the (irrelevant) read of y to read 0, we obtain an SC-consistent graph that is observationally equivalent to G.

Next, to be able to check for observational robustness of a given program, we present *non-observational-robustness witnesses* which, as before, allow us to verify observational robustness by considering only SC-executions of the program. For this definition, we use *tainted* versions of several relations w.r.t. a set $T \subseteq G.\mathsf{IR}$ of G-irrelevant reads (see Def. 3.14), defined as follows:

Definition 4.13. Given $T \subseteq G.\mathsf{IR}$, the T-*tainted* versions of $\mathtt{rf}$, $\mathtt{sw}$, and hb are defined as follows:[4]

$$G.\mathtt{rf}_{\overline{T}} \triangleq G.\mathtt{rf} \,;\, [\mathsf{Event} \setminus T]$$

$$G.\mathtt{sw}_{\overline{T}} \triangleq [G.\mathsf{E}^{\sqsupseteq \mathsf{rel}}] \,;\, ([G.\mathsf{F}] \,;\, G.\mathsf{po})^{?} \,;\, G.\mathtt{rf}^{+}_{\overline{T}} \,;\, (G.\mathsf{po} \,;\, [G.\mathsf{F}])^{?} \,;\, [G.\mathsf{E}^{\sqsupseteq \mathsf{acq}}]$$

$$G.\mathsf{hb}_{\overline{T}} \triangleq (G.\mathsf{po} \cup G.\mathtt{sw}_{\overline{T}})^{+}$$

Definition 4.14. A *non-observational-robustness witness* for a program *Pr* is a tuple $\langle \overline{q}, G, \tau, L, \psi, T \rangle$ such that the following hold:

- G is generated by *Pr* with final state $\overline{q}$ and dependency store ψ.
- G is SC-consistent.
- $\overline{q}$ enables the transition $\langle \tau, L \rangle$.
- $\mathtt{typ}(L) \in \{\mathtt{W}, \mathtt{RMW}, \mathtt{R}^{\star}, \mathtt{CTRL}\}$.
- $T \subseteq G.\mathsf{IR}$.
- $G.\mathtt{rf} \,;\, [T] \subseteq G.\mathsf{hb}_{\mathsf{SC}} \,;\, G.\mathsf{po} \setminus G.\mathtt{rf}^{?}_{\overline{T}} \,;\, G.\mathsf{hb}_{\overline{T}}$.
- Either $T \cap \psi(\mathsf{R}_{\mathsf{in}}(L)) \neq \emptyset$ or the following hold:
  - $\mathtt{typ}(L) \in \{\mathtt{W}, \mathtt{RMW}, \mathtt{R}^{\star}\}$.
  - $G.w^{\mathsf{max}}_{\mathtt{loc}(L)} \in \mathit{dom}(G.\mathsf{hb}^{?}_{\mathsf{SC}} \,;\, [\mathsf{E}^{\tau}])$.
  - The following hold for some $w \in G.\mathsf{W}_{\mathtt{loc}(L)}$:
    * $w \neq G.w^{\mathsf{max}}_{\mathtt{loc}(L)}$.
    * $w \notin \mathit{dom}(G.\mathtt{mo} \,;\, G.\mathtt{rf}^{?}_{\overline{T}} \,;\, G.\mathsf{hb}^{?}_{\overline{T}} \,;\, [\mathsf{E}^{\tau}])$.
    * If $\mathtt{typ}(L) \in \{\mathtt{W}, \mathtt{RMW}\}$, then $w \notin \mathit{dom}(G.\mathtt{rf} \,;\, [G.\mathsf{RMW}])$.
    * If $\mathtt{typ}(L) \in \{\mathtt{RMW}, \mathtt{R}^{\star}\}$, then $G.\mathtt{val}_{\mathtt{W}}(w) = \mathtt{val}_{\mathtt{R}}(L)$.

Non-observational-robustness witnesses are similar to non-robustness witnesses (Def. 4.5), but they also contain a dependency store ψ and a set T of "tainted" read events in G. The events in T are the read events in G that could have read from an $\mathtt{mo}$-earlier write event (w.r.t. their current $\mathtt{rf}$-source) under RC20 but not under SC (i.e., $G.\mathtt{rf} \,;\, [T] \subseteq G.\mathsf{hb}_{\mathsf{SC}} \,;\, G.\mathsf{po} \setminus G.\mathtt{rf}^{?}_{\overline{T}} \,;\, G.\mathsf{hb}_{\overline{T}}$.)

[4]By definition, T cannot contain events in $G.\mathsf{RMW}$, so we always have $G.\mathtt{rf}^{+}_{\overline{T}} = G.\mathtt{rf}^{*} \,;\, G.\mathtt{rf}_{\overline{T}}$.

The existence of such a read event would entail a robustness violation, but *not* an *observational* robustness violation, which may be violated only if some later event (in particular, a CTRL event) depends on an event in T (i.e., $T \cap \psi(\mathsf{R_{in}}(L)) \neq \emptyset$). When $\mathtt{typ}(L) \in \{\mathtt{W}, \mathtt{RMW}, \mathtt{R}^\star\}$, we have the same conditions as in Def. 4.5, but crucially, the read-from edges in G to tainted read events should not limit the writes that threads can observe under RC20. Hence, we use the tainted versions of $\mathtt{rf}$ and hb in the $w \notin dom(G.\mathsf{mo} \,;G.\mathtt{rf}^?_{\overline{T}} \,; G.\mathsf{hb}^?_{\overline{T}} \,; [\mathsf{E}^\tau])$ condition.

Example 4.15. A non-observational-robustness witness for the (non-fenced) MP program in Ex. 4.2 consists of the graph depicted on the right in that example, where the set T includes the highlighted event reading $\mathtt{x}$. The read event in T could have read from the initial write under RC20 but not under SC; and the program enables a transition that depends on the event in T (executing the $\mathtt{assert(b} \neq 0\mathtt{)}$).

Next, we establish that non-observational-robustness witnesses must exist for non-observationally-robust programs. The proof is rather intricate DRF-style argument—RC20 can deviate from SC early on, and to construct the witness for a non-robust program we need to globally "fix" these deviations.

THEOREM 4.16. *If Pr is not observationally robust, then there exists a non-observational-robustness witness for Pr.*

The following lemma is a key lemma in the proof of Thm. 4.16.

LEMMA 4.17. *Let G be an* RC20*-consistent execution graph. Suppose that $G.\mathsf{hb}^{\mathsf{dp}}_{\mathsf{SC}}$ is irreflexive. Then, there exist an* SC*-consistent execution graph G' and a set $T \subseteq G.\mathsf{IR}$ such that $G \sim_T G'$ and $G'.\mathtt{rf} \,; [T] \subseteq G'.\mathsf{hb}_{\mathsf{SC}} \,; G'.\mathsf{po} \setminus G'.\mathtt{rf}^?_{\overline{T}} \,; G'.\mathsf{hb}_{\overline{T}}$.*

PROOF (OUTLINE). Let S be a maximal acyclic relation such that

$$G.\mathsf{po} \cup G.\mathtt{rf} \cup G.\mathsf{mo} \cup [dom(G.\mathsf{dp}) \cup G.\mathsf{R}^\star] \,; G.\mathtt{fr} \subseteq S \subseteq G.\mathsf{po} \cup G.\mathtt{rf} \cup G.\mathsf{mo} \cup G.\mathtt{fr}.$$

Define the following:

- For every $r \in G.\mathsf{R}$, let $W_r = \{w \mid \langle r, w\rangle \in G.\mathtt{fr} \setminus S^+\}$.
- Let $T = \{r \in G.\mathsf{R} \mid W_r \neq \emptyset\}$.
- For every $r \in T$, let $w_r = \max_{G.\mathsf{mo}} W_r$.

Then, one shows that the following properties hold:

- $T \subseteq G.\mathsf{IR}$.
- If $\langle w, r\rangle \in G.\mathtt{rf} \,; [T]$, then $\langle w, w_r\rangle \in G.\mathsf{mo}$.
- $\langle w_r, r\rangle \in S^+ \,; G.\mathsf{po}$ for every $r \in T$.
- $\langle r, w\rangle \in S^+$ for every $r \in T$ and w such that $\langle w_r, w\rangle \in G.\mathsf{mo}$.

Now, we let G' be the execution graph obtained from G by modifying each event $r \in T$ to read from w_r (and adapting read values accordingly). Our construction ensures that $G \sim_T G'$, $G.\mathtt{rf} \subseteq (G'.\mathsf{mo})^? \,; G'.\mathtt{rf}$, $G.\mathtt{fr} \cap S \subseteq G'.\mathtt{fr}$, and $S \subseteq G'.\mathsf{hb}_{\mathsf{SC}}$. With these properties, we can show that the two required conditions hold. □

Next, we outline the proof of Thm. 4.16.

PROOF OF THM. 4.16 (OUTLINE). Let $\mathcal{G}$ be the set consisting of all execution graphs G such that (i) G is RC20-consistent; (ii) G is generated by Pr; and (iii) $G.\mathsf{hb}^{\mathsf{dp}}_{\mathsf{SC}}$ is not irreflexive. Since Pr is not observationally robust, $\mathcal{G}$ is not empty. Let G_1 be a minimal element in $\mathcal{G}$, in the sense that every proper prefix of G_1 is not in $\mathcal{G}$. Let $\overline{q}_1 \in Pr.\mathsf{Q}$ and $\psi_1 : \mathsf{Reg} \to \mathcal{P}(\mathsf{Event})$ such that $\langle \overline{q}_1, G_1, \psi_1\rangle$ is reachable in $Pr \parallel \mathsf{FG}$. Let $G_2 \in \mathsf{EGraph}$, $\overline{q}_2 \in Pr.\mathsf{Q}$, $\psi_2 : \mathsf{Reg} \to \mathcal{P}(\mathsf{Event})$, $\tau \in \mathsf{Tid}$, and $L = \langle l, R_{\mathsf{in}}, R_{\mathsf{out}}\rangle \in \mathsf{TLab}$ such that:

- $l \in \mathsf{Lab}$.
- $\langle \overline{q}_2, G_2, \psi_2 \rangle$ is reachable in $Pr \parallel \mathsf{FG}$.
- $\overline{q}_2 \xrightarrow{\tau,L}_{Pr} \overline{q}$ for some $\overline{q} \in Pr.\mathsf{Q}$ such that $\overline{q} \xrightarrow{\_,\langle \epsilon,\_,\_\rangle}{}^{*}_{Pr} \overline{q}_1$.
- $\langle G_2, \psi_2 \rangle \xrightarrow{\tau,L}_{\mathsf{FG}} \langle G_1, \psi_1 \rangle$.

Then, by definition, there exists some $w_c \in \mathsf{Event} \cup \{\bot\}$ such that the following hold:

- $G_1 = \mathtt{add}(G_2, \tau, l, \psi_2(R_{\mathsf{in}}), w_c)$.
- If $\mathtt{typ}(l) \in \{\mathtt{R}, \mathtt{RMW}, \mathtt{R}^\star\}$, then $G_2.\mathtt{val}_{\mathtt{W}}(w_c) = \mathtt{val}_{\mathtt{R}}(l)$.
- If $\mathtt{typ}(l) \notin \{\mathtt{F}, \mathtt{CTRL}\}$, then $w_c \in G_2.\mathsf{W}_{\mathtt{loc}(l)}$.
- If $\mathtt{typ}(l) \in \{\mathtt{F}, \mathtt{CTRL}\}$, then $w_c = \bot$.

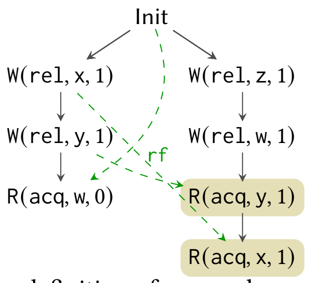
Let $c = \mathsf{next}(G_2, \tau)$ (i.e., the event added to G_2 to obtain G_1). Then, $G_2 = G_1 \setminus \{c\}$ is RC20-consistent, and the minimality of G_1 ensures that $G_2.\mathsf{hb}^{\mathsf{dp}}_{\mathsf{SC}}$ is irreflexive.

By Lemma 4.17 there exist an SC-consistent execution graph G_3 and a set $T \subseteq G_2.\mathsf{IR}$ such that $G_2 \sim_T G_3$ and $G_3.\mathtt{rf} ; [T] \subseteq G_3.\mathsf{hb}_{\mathsf{SC}} ; G_3.\mathsf{po} \setminus G_3.\mathtt{rf}^?_{\overline{T}} ; G_3.\mathsf{hb}_{\overline{T}}$. Since $T \subseteq G_2.\mathsf{IR}$, Lemma 3.17 implies that $\langle \overline{q}_3, G_3, \psi_2 \rangle$ is reachable in $Pr \parallel \mathsf{FG}$ for some state $\overline{q}_3 \in Pr.\mathsf{Q}$ such that $\overline{q}_2 \sim_R \overline{q}_3$ for $R = \{r \in \mathsf{Reg} \mid \psi_2(r) \cap T \neq \emptyset\}$. We also have that $\overline{q}_3$ enables the transition $\langle \tau, L \rangle$ if $\psi_2(R_{\mathsf{in}}) \cap T = \emptyset$; and that $\overline{q}_3$ enables $\langle \tau, \langle l_3, R_{\mathsf{in}}, R_{\mathsf{out}} \rangle \rangle$ for some event label $l_3 \in \mathsf{Lab}$.

Now, if $\psi_2(R_{\mathsf{in}}) \cap T \neq \emptyset$, then $\langle \overline{q}_3, G_3, \tau, \langle l_3, R_{\mathsf{in}}, R_{\mathsf{out}} \rangle, \psi_2, T \rangle$ is a non-observational-robustness witness for Pr. Otherwise, we can show that $\mathtt{typ}(l) \in \{\mathtt{W}, \mathtt{RMW}, \mathtt{R}^\star\}$ and $\langle c, c \rangle \in (G_1.\mathtt{mo} \cup G_1.\mathtt{fr}) ; G_1.\mathsf{hb}^{\mathsf{dp}}_{\mathsf{SC}} ; G_1.\mathsf{po}$. Let $E_4 = dom(G_3.\mathsf{hb}_{\mathsf{SC}} ; [\mathsf{E}^\tau])$, $G_4 = G_3 \cap E_4$ and $T' = T \cap E_4$. Since G_4 is a prefix of G_3 and $G_3.\mathsf{E}^\tau \subseteq G_4.\mathsf{E}$, there exist $\overline{q}_4 \in Pr.\mathsf{Q}$ and dependency store ψ_4, such that $\langle \overline{q}_4, G_4, \psi_4 \rangle$ is reachable in $Pr \parallel \mathsf{FG}$ and $\overline{q}_4$ enables $\langle \tau, L \rangle$. As a prefix of G_3, G_4 is SC-consitent, and since $G_3.\mathtt{rf} ; [T] \subseteq G_3.\mathsf{hb}_{\mathsf{SC}} ; G_3.\mathsf{po} \setminus G_3.\mathtt{rf}^?_{\overline{T}} ; G_3.\mathsf{hb}_{\overline{T}}$, our construction ensures that $G_4.\mathtt{rf} ; [T'] \subseteq G_4.\mathsf{hb}_{\mathsf{SC}} ; G_4.\mathsf{po} \setminus G_4.\mathtt{rf}^?_{\overline{T'}} ; G_4.\mathsf{hb}_{\overline{T'}}$. It follows that $\langle \overline{q}_4, G_4, \tau, L, \psi_4, T' \rangle$ is a non-observational-robustness witness for Pr. □

Example 4.18. An important detail in the definition of a non-observational-robustness witness is the fact that we use $G.\mathtt{rf}^?_{\overline{T}} ; G.\mathsf{hb}^?_{\overline{T}}$ rather than $G.\mathtt{rf}^? ; G.\mathsf{hb}^?$ in the condition checking that thread τ is not "aware" of a later write than w. Intuitively speaking, since the execution graph in the witness is SC-consistent, the taint reads in this graph also read the most recent values, but these reads must not make the thread aware of more writes, since they could have read from an $\mathtt{mo}$-earlier write under RC20. The following example of a "nested" SB litmus test demonstrates this issue. Under RC20 (or any other weak memory model), we may get the non-SC outcome $\mathtt{b} = \mathtt{d} = 0$, which, because of the conditionals, implies that the program is not observationally robust. The graph component of a possible non-observational-robustness witness is presented one the right (all initialization event summarized in one event and $\mathtt{mo}$-edges are omitted) where the highlighted events are the tainted read events in the set T.

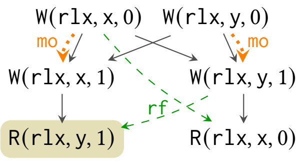
```
x.store(1, rel)          || z.store(1, rel)
y.store(1, rel)          || w.store(1, rel)
a := w.load(acq)         || c := y.load(acq)
b := z.load(acq)         || d := x.load(acq)
if b = 0 goto ...        || if d = 0 goto ...
```



However, if we use the non-tainted versions of $\mathtt{rf}$ and hb in the definition of a non-observational-robustness witness, we will not be able to include the read from $\mathtt{x}$ in T (since we will wrongly have that the thread is "aware" of $\mathtt{W}(\mathtt{rel}, \mathtt{x}, 1)$ and cannot read the initial value 0). In fact, with $G.\mathtt{rf}^? \,; G.\mathtt{hb}^?$ instead of $G.\mathtt{rf}^?_{\overline{T}} \,; G.\mathtt{hb}^?_{\overline{T}}$, we will not have *any* non-observational-robustness witness for this non-observationally-robust example.

Example 4.19. The converse of the claim in Thm. 4.16 does not hold. The following simple adaptation of the SB litmus test is an observationally robust program that has an non-observational-robustness witness (whose execution graph component is presented on the right, where the highlighted event is the only event in T). Developing an exact (and verifiable) characterization of observational-robustness is challenging and is left for future work.

```
x.store(1, rlx)       ‖ y.store(1, rlx)
a := y.load(rlx)      ‖ b := x.load(rlx)
assert(a = a)         ‖
```



Finally, by adding (vacuous) assert instructions we can reduce robustness verification to the existence of a non-observational-robustness witness.

Lemma 4.20. *Let Pr be a program in which every read instruction* $r := x.\mathtt{load}(o_\mathsf{R})$ *is followed by a (vacuous) assert instruction* $\mathtt{assert}(r = r)$*. Then, there exists a non-robustness witness for Pr iff there exists a non-observational-robustness witness for Pr.*

Using Lemma 4.20, we obtain a reduction from robustness verification to the problem of verifying whether a non-observational-robustness witness exists or not. In the rest of this paper we focus on the latter problem, and obtain a decision procedure for robustness using this reduction.

5 VERIFYING OBSERVATIONAL ROBUSTNESS

In this section we show how to automatically check for the existence of a non-observational-robustness witness by exploring the reachable states in an instrumented SC semantics. We present the instrumented semantics in two stages. First, §5.1 presents an "operational" version of Thm. 4.16 in the form of an (infinite) memory system that we call $\mathsf{SC_T}$. This system extends the standard operational construction of SC-consistent execution graphs with instrumentation and monitoring for non-observational-robustness witnesses. Then, in §5.2 we present $\mathsf{SC_M}$, a finite memory system instrumenting the standard SC memory system, and show that it simulates $\mathsf{SC_T}$. Finally, we obtain that non-observational-robustness witness exists iff certain error state is reachable under $\mathsf{SC_M}$.

5.1 The Memory System $\mathsf{SC_T}$

The memory system $\mathsf{SC_T}$ provides an "operational" version of Thm. 4.16. It is based on an operationalized version of the declarative SC semantics (see Def. 3.7), which restricts the steps of the graph generation system FG (Def. 3.10) so that reads only read from the last executed write (to the relevant location) and that writes are always placed in the end of the (per-location) mo-order. In addition, $\mathsf{SC_T}$ includes instrumentation and monitoring that (i) tracks a "taint set" T of read events that read stale values during the execution; and (ii) enters an error state, denoted by $\perp$, when a non-observational-robustness witness is detected.

Definition 5.1. The memory system $\mathsf{SC_T}$ is given by:

- Its states consists of the error state $\perp$, and all tuples of the form $\langle G, \psi, T\rangle$, where G is an execution graph, ψ is a dependency store, and $T \subseteq \{e \in G.\mathsf{E} \mid G.\mathtt{typ}(e) = \mathsf{R}\}$.
- Its initial state is $\langle G_0, \psi_0, \emptyset\rangle$ (where G_0 and ψ_0 are defined in Def. 3.10).
- Its transitions are given in Fig. 4.

$$\textsc{silent/fence/control}\quad \frac{\langle G,\psi\rangle \xrightarrow{\tau,L}_{\mathsf{FG}} \langle G',\psi'\rangle \qquad \mathtt{lab}(L)=\varepsilon \lor \mathtt{typ}(L)\in\{\mathtt{F},\mathtt{CTRL}\}}{\langle G,\psi,T\rangle \xrightarrow{\tau,L}_{\mathsf{SC_T}} \langle G',\psi',T\rangle}$$

$$\textsc{non-read}\quad \frac{\langle G,\psi\rangle \xrightarrow{\tau,L}_{\mathsf{FG}} \langle G',\psi'\rangle \qquad \mathtt{typ}(L)\in\{\mathtt{W},\mathtt{RMW},\mathtt{R}^{\star}\} \qquad G'=\mathtt{add}(G,\tau,\_,\_,G.w^{\max}_{\mathtt{loc}(L)})}{\langle G,\psi,T\rangle \xrightarrow{\tau,L}_{\mathsf{SC_T}} \langle G',\psi',T\rangle}$$

$$\textsc{read}\quad \frac{\langle G,\psi\rangle \xrightarrow{\tau,L}_{\mathsf{FG}} \langle G',\psi'\rangle \qquad \mathtt{typ}(L)=\mathtt{R} \qquad G'=\mathtt{add}(G,\tau,\_,\_,G.w^{\max}_{\mathtt{loc}(L)}) \\ G.w^{\max}_{\mathtt{loc}(L)}\in dom(G.\mathtt{hb}_{\mathsf{SC}}\,;[\mathsf{E}^{\tau}]) \implies G.w^{\max}_{\mathtt{loc}(L)}\in dom(G.\mathtt{rf}^{?}_{\overline{T}}\,;G.\mathtt{hb}_{\overline{T}}\,;[\mathsf{E}^{\tau}])}{\langle G,\psi,T\rangle \xrightarrow{\tau,L}_{\mathsf{SC_T}} \langle G',\psi',T\rangle}$$

$$\textsc{err-dep}\quad \frac{\mathtt{typ}(L)\in\{\mathtt{W},\mathtt{RMW},\mathtt{R}^{\star},\mathtt{CTRL}\} \\ \psi(\mathsf{R}_{\mathsf{in}}(L))\cap T\neq\emptyset}{\langle G,\psi,T\rangle \xrightarrow{\tau,L}_{\mathsf{SC_T}} \bot}$$

$$\textsc{taint-read}\quad \frac{\langle G,\psi\rangle \xrightarrow{\tau,L}_{\mathsf{FG}} \langle G',\psi'\rangle \qquad \mathtt{typ}(L)=\mathtt{R} \\ G'=\mathtt{add}(G,\tau,\_,\_,G.w^{\max}_{\mathtt{loc}(L)}) \\ T'=T\cup\{\mathtt{next}(G,\tau)\} \\ G.w^{\max}_{\mathtt{loc}(L)}\in dom(G.\mathtt{hb}_{\mathsf{SC}}\,;[\mathsf{E}^{\tau}]) \\ G.w^{\max}_{\mathtt{loc}(L)}\notin dom(G.\mathtt{rf}^{?}_{\overline{T}}\,;G.\mathtt{hb}_{\overline{T}}\,;[\mathsf{E}^{\tau}])}{\langle G,\psi,T\rangle \xrightarrow{\tau,L}_{\mathsf{SC_T}} \langle G',\psi',T'\rangle}$$

$$\textsc{err-sc}\quad \frac{\mathtt{typ}(L)\in\{\mathtt{W},\mathtt{RMW},\mathtt{R}^{\star}\} \\ G.w^{\max}_{\mathtt{loc}(L)}\in dom(G.\mathtt{hb}_{\mathsf{SC}}\,;[\mathsf{E}^{\tau}]) \qquad w\neq G.w^{\max}_{\mathtt{loc}(L)} \\ w\in G.\mathsf{W}_{\mathtt{loc}(L)} \qquad w\notin dom(G.\mathtt{mo}\,;G.\mathtt{rf}^{?}_{\overline{T}}\,;G.\mathtt{hb}^{?}_{\overline{T}}\,;[\mathsf{E}^{\tau}]) \\ \mathtt{typ}(L)\in\{\mathtt{W},\mathtt{RMW}\} \implies w\notin dom(G.\mathtt{rf}\,;[G.\mathsf{RMW}]) \\ \mathtt{typ}(L)\in\{\mathtt{RMW},\mathtt{R}^{\star}\} \implies G.\mathtt{val}_{\mathsf{W}}(w)=\mathtt{val}_{\mathsf{R}}(L)}{\langle G,\psi,T\rangle \xrightarrow{\tau,L}_{\mathsf{SC_T}} \bot}$$

Fig. 4. $\mathsf{SC_T}$ transitions.

Each of the (non-error) steps of $\mathsf{SC_T}$ extends the current graph with one event (like the FG system in Def. 3.10) while recording in the set T all the read events added to the graph that read taint values, maintaining the set T to consist of all the tainted read events in the current graph. Exactly as in the definition of non-observational-robustness witness, the latter formally means that there is a $\mathtt{hb}_{\mathsf{SC}}$-path, but not an $\mathtt{rf}^{?}_{\overline{T}}\,;\mathtt{hb}_{\overline{T}}$-path from the $\mathtt{mo}$-maximal write to the relevant location to the thread executing the read. In addition, two steps lead the system to the error state when a non-observational-robustness witness is detected. The soundness and completeness of $\mathsf{SC_T}$ is established in the following theorem.

Theorem 5.2. *There exists a non-observational-robustness witness for a program Pr iff some state of the form $\langle\overline{q},\bot\rangle$ is reachable in $Pr \parallel \mathsf{SC_T}$.*

5.2 The Memory System $\mathsf{SC_M}$

We show how to automatically check whether a state of the form $\langle\_,\bot\rangle$ is reachable in $Pr \parallel \mathsf{SC_T}$. Since execution graphs (for programs with loops) may grow unboundedly, $\mathsf{SC_T}$ is an infinite state transition system whose traces cannot be naively explored. To address this, we show that maintaining the whole execution graph that was generated so far, as done in $\mathsf{SC_T}$, is unnecessary. Instead, it is possible to summarize $\mathsf{SC_T}$'s state and record only the properties of $\langle G,\psi,T\rangle$ that are needed for checking which transitions are enabled in each state and whether a transition to the error state is enabled or not. To do so, we define a *finite* (but precise) abstraction of $\mathsf{SC_T}$, a memory system which we call $\mathsf{SC_M}$ (for SC with Monitors), that simulates $\mathsf{SC_T}$, so that the traces of $\mathsf{SC_T}$ that reach the error state coincide with those of $\mathsf{SC_M}$ that reach the error state.

Next, we present $\mathsf{SC_M}$'s states and the transitions between them. We write $q_{\mathsf{SC_M}}(G,\psi,T)$ for the $\mathsf{SC_M}$ state that abstracts the $\mathsf{SC_T}$ state $\langle G,\psi,T\rangle$. States of $\mathsf{SC_M}$ are tuples of the form $q_{\mathsf{SC_M}} =$

$\langle M, R_{\mathsf{taint}}, V_{\mathsf{SC}}, V\rangle$. We use $q_{\mathsf{SC_M}}.\mathsf{M}$, $q_{\mathsf{SC_M}}.\mathsf{R_{taint}}$, $q_{\mathsf{SC_M}}.\mathsf{V_{SC}}$, and $q_{\mathsf{SC_M}}.\mathsf{V}$ to refer to the different components. Below, we gradually present the structure and meaning of each component and describe how it is maintained in the system's transitions.

Memory (M). Since $\mathsf{SC_T}$ monitors SC-*executions*, for checking what transitions are enabled (whether $\langle G, \psi, T\rangle$ enables a transition $\langle \tau, L\rangle$ in $\mathsf{SC_T}$), we only need to track the last value written to every location. This Loc $\rightarrow$ Val mapping is stored in the M component. Formally, we have

$$q_{\mathsf{SC_M}}(G, \psi, T).\mathsf{M} = \lambda x.\, G.\mathtt{val_W}(G.w_x^{\max}).$$

Then, $\langle \tau, L\rangle$ is enabled iff $\mathtt{typ}(L) \in \{\mathtt{W}, \mathtt{F}, \mathtt{CTRL}\}$ (these are always enabled by the memory) or $\mathtt{typ}(L) \in \{\mathsf{R}, \mathsf{RMW}, \mathsf{R}^\star\}$ and $\mathtt{val_R}(L) = q_{\mathsf{SC_M}}.\mathsf{M}(\mathtt{loc}(L))$. (The rest of the system only instruments and monitors the runs, so it does not affect the transitions between non-error states.) Initially, we have $M_\emptyset = \lambda x.\, 0$, since the initial execution graph contains initialization to 0 for all locations. The maintenance of M is trivial: write and RMW steps to x update the value of x, while other transitions keep the memory intact.

Tainted Registers (R_{taint}). Instead of maintaining the (unbounded) set T of tainted events, $\mathsf{SC_M}$ keeps a (bounded) set $R_{\mathsf{taint}} \subseteq \mathsf{Reg}$ of *tainted registers*. A register is *tainted* if its value depends on some event $e \in T$. Formally, we have

$$q_{\mathsf{SC_M}}(G, \psi, T).\mathsf{R_{taint}} = \{r \in \mathsf{Reg} \mid \psi(r) \cap T \neq \emptyset\}.$$

Then, we replace the condition $\psi(\mathsf{R_{in}}(L)) \cap T \neq \emptyset$ in $\mathsf{SC_T}$'s ERR-DEP step with $\mathsf{R_{in}}(L) \cap R_{\mathsf{taint}} \neq \emptyset$. Initially, we have $R_{\mathsf{taint}\emptyset} = \emptyset$. Registers are added to R_{taint} when a taint value is loaded to them (in a read instruction), or if any expression that includes tainted registers is assigned to them. Registers are removed from R_{taint} when they are overwritten with a non-taint value (e.g., by another read instruction). Whether a read transition reads a taint value or not (i.e., whether RC20 allows to read from some other event in the graph) is detected using the components described below.

$\mathsf{hb_{SC}}$*-tracker* (V_{SC}). The V_{SC} component in $\mathsf{SC_M}$ states is used to track whether a thread is "$\mathsf{hb_{SC}}$-aware" of the maximal write to some location. This piece of information is needed for checking the $G.w_{\mathtt{loc}(L)}^{\max} \in \mathit{dom}(G.\mathsf{hb_{SC}} \,;\, [\mathsf{E}^\tau])$ condition that appears in $\mathsf{SC_T}$'s READ, TAINT-READ, and ERR-SC steps. The V_{SC} component is defined and maintained exactly as in [Lahav and Margalit 2019]. To save space, we omit the details here and simply assume transitions of the form $V_{\mathsf{SC}} \xrightarrow{\tau, l} V'_{\mathsf{SC}}$ (where $\tau \in \mathsf{Tid}$ and $l \in \mathsf{Lab}$) for maintaining this component, and a query of the form $x \in V_{\mathsf{SC}}(\tau)$ that "returns true" iff $G.w_x^{\max} \in \mathit{dom}(G.\mathsf{hb}_{\mathsf{SC}}^? \,;\, [\mathsf{E}^\tau])$. Initially, we start with $V_{\mathsf{SC}\emptyset}$, where $x \in V_{\mathsf{SC}\emptyset}(\tau)$ always "returns true". We refer the reader to [Lahav and Margalit 2019, §5] for details.

RC20 *tracker* (V). The V component in $\mathsf{SC_M}$ states is the most complex component in $\mathsf{SC_M}$'s states. It provides the additional information required for detecting robustness violations, as well as for differentiating between normal and taint read steps (for maintaining $q_{\mathsf{SC_M}}.R_{\mathsf{taint}}$). First, we need to track for every thread τ and location x, whether, besides the mo-maximal write, there exists a write w to x in the current execution graph that does not have an-$\mathsf{mo} \,;\, \mathsf{rf}^?_{\overline{T}} \,;\, \mathsf{hb}^?_{\overline{T}}$-path to some event of thread τ (which means that w can be observed by thread τ under RC20). Indeed, this information is needed for checking the $G.w_{\mathtt{loc}(L)}^{\max} \in \mathit{dom}(G.\mathsf{rf}^?_{\overline{T}} \,;\, G.\mathsf{hb}_{\overline{T}} \,;\, [\mathsf{E}^\tau])$ condition in $\mathsf{SC_T}$'s READ and TAINT-READ steps, as well as the $w \notin \mathit{dom}(G.\mathsf{mo} \,;\, G.\mathsf{rf}^?_{\overline{T}} \,;\, G.\mathsf{hb}^?_{\overline{T}} \,;\, [\mathsf{E}^\tau])$ condition in $\mathsf{SC_T}$'s ERR-SC step. Furthermore, for $\mathsf{SC_T}$'s ERR-SC step, we also need to know:

- whether w is already read by some RMW event in the graph ($w \notin \mathit{dom}(\mathsf{rf} \,;\, [\mathsf{RMW}])$), in which case we cannot add a write placed as the mo-successor of w without violating ATOMICITY; and

- whether w's written value is a value that can be read by the program when adding a certain label l, since for CAS and blocking instructions (BCAS and wait), the program only allows to add $\mathsf{R}^\star$ or RMW events that read certain values.

We achieve this by maintaining a tuple $V = \langle V^{\mathsf{a}}, V^{\mathsf{c}}, V^{\mathsf{r}}, V^{\mathsf{w}}, V^{\mathsf{a}}_{\mathsf{RMW}}, V^{\mathsf{c}}_{\mathsf{RMW}}, V^{\mathsf{r}}_{\mathsf{RMW}}, V^{\mathsf{w}}_{\mathsf{RMW}} \rangle$, where: $V^{\alpha}, V^{\alpha}_{\mathsf{RMW}} : \mathsf{Tid} \times \mathsf{Loc} \to \mathcal{P}(\mathsf{Val})$ for $\alpha \in \{\mathsf{a}, \mathsf{c}, \mathsf{r}\}$, and $V^{\mathsf{w}}, V^{\mathsf{w}}_{\mathsf{RMW}} : \mathsf{Loc} \times \mathsf{Loc} \to \mathcal{P}(\mathsf{Val})$. The V^{c} and $V^{\mathsf{c}}_{\mathsf{RMW}}$ components (standing for *current view* and *current RMW-view*) are the ones actually used for tracking the information mentioned above: we have $v \in V^{\mathsf{c}}(\tau, x)$ iff the value v is written by some write w to x in the current execution graph such that $w \neq w^{\max}_x$ and $w \notin dom(\mathsf{mo} ; \mathsf{rf}^{?}_{\overline{T}} ; \mathsf{hb}^{?}_{\overline{T}} ; [\mathsf{E}^{\tau}])$; and we have $v \in V^{\mathsf{c}}_{\mathsf{RMW}}(\tau, x)$ iff $w \notin dom(\mathsf{rf} ; [\mathsf{RMW}])$ in addition to the above conditions. The other six components provide the additional instrumentation required for a faithful maintenance of V^{c} and $V^{\mathsf{c}}_{\mathsf{RMW}}$. Roughly speaking:

- V^{a} and $V^{\mathsf{a}}_{\mathsf{RMW}}$ (*acquire view* and *acquire RMW-view*) are needed to handle acquire fences.
- V^{r} and $V^{\mathsf{r}}_{\mathsf{RMW}}$ (*release view* and *release RMW-view*) are needed to handle release fences.
- V^{w} and $V^{\mathsf{w}}_{\mathsf{RMW}}$ (*location view* and *location RMW-view*) are needed to handle reads and RMWs, which incorporate the view of the write event they read from in the thread view. (Crucially, since we are running under SC, we only need to record this data for mo-maximal write events.)

We use $q_{\mathsf{SC}_\mathsf{M}}.\mathsf{V}^{\alpha}$ and $q_{\mathsf{SC}_\mathsf{M}}.\mathsf{V}^{\alpha}_{\mathsf{RMW}}$ (for $\alpha \in \{\mathsf{a}, \mathsf{c}, \mathsf{r}, \mathsf{w}\}$) to directly accesses these views. Their formal meaning is given by ($G.\mathtt{val}_{\mathsf{W}}$ is lifted to sets of events in the obvious way):

$$
\begin{aligned}
q_{\mathsf{SC}_\mathsf{M}}(G, \psi, T).\mathsf{V}^{\mathsf{c}} &= \lambda \tau, x.\, G.\mathtt{val}_{\mathsf{W}}[G.\mathsf{W}^{\neq\max}_x \setminus dom(R ; [\mathsf{E}^{\tau}])] \\
q_{\mathsf{SC}_\mathsf{M}}(G, \psi, T).\mathsf{V}^{\mathsf{r}} &= \lambda \tau, x.\, G.\mathtt{val}_{\mathsf{W}}[G.\mathsf{W}^{\neq\max}_x \setminus dom(R ; [\mathsf{E}^{\tau} \cap G.\mathsf{F}^{\sqsupseteq\mathtt{rel}}])] \\
q_{\mathsf{SC}_\mathsf{M}}(G, \psi, T).\mathsf{V}^{\mathsf{a}} &= \lambda \tau, x.\, G.\mathtt{val}_{\mathsf{W}}[G.\mathsf{W}^{\neq\max}_x \setminus dom(R ; ([G.\mathsf{E}^{\sqsupseteq\mathtt{rel}}]; ([G.\mathsf{F}]; G.\mathsf{po})^{?}; G.\mathsf{rf}^{+}_{\overline{T}})^{?} ; [\mathsf{E}^{\tau}]) \\
q_{\mathsf{SC}_\mathsf{M}}(G, \psi, T).\mathsf{V}^{\mathsf{w}} &= \lambda y, x.\, G.\mathtt{val}_{\mathsf{W}}[G.\mathsf{W}^{\neq\max}_x \setminus dom(R ; [G.\mathsf{E}^{\sqsupseteq\mathtt{rel}}]; ([G.\mathsf{F}]; G.\mathsf{po})^{?}; G.\mathsf{rf}^{*}_{\overline{T}} ; [G.w^{\max}_y]) \\
q_{\mathsf{SC}_\mathsf{M}}(G, \psi, T).\mathsf{V}^{\mathsf{c}}_{\mathsf{RMW}} &= \lambda \tau, x.\, G.\mathtt{val}_{\mathsf{W}}[G.\mathsf{W}^{\neq\max}_x \setminus dom(R ; [\mathsf{E}^{\tau}] \cup R_{\mathsf{RMW}})] \\
q_{\mathsf{SC}_\mathsf{M}}(G, \psi, T).\mathsf{V}^{\mathsf{r}}_{\mathsf{RMW}} &= \lambda \tau, x.\, G.\mathtt{val}_{\mathsf{W}}[G.\mathsf{W}^{\neq\max}_x \setminus dom(R ; [\mathsf{E}^{\tau} \cap G.\mathsf{F}^{\sqsupseteq\mathtt{rel}}] \cup R_{\mathsf{RMW}})] \\
q_{\mathsf{SC}_\mathsf{M}}(G, \psi, T).\mathsf{V}^{\mathsf{a}}_{\mathsf{RMW}} &= \lambda \tau, x.\, G.\mathtt{val}_{\mathsf{W}}[G.\mathsf{W}^{\neq\max}_x \setminus dom(R ; ([G.\mathsf{E}^{\sqsupseteq\mathtt{rel}}]; ([G.\mathsf{F}]; G.\mathsf{po})^{?}; G.\mathsf{rf}^{+}_{\overline{T}})^{?} ; [\mathsf{E}^{\tau}] \cup R_{\mathsf{RMW}}) \\
q_{\mathsf{SC}_\mathsf{M}}(G, \psi, T).\mathsf{V}^{\mathsf{w}}_{\mathsf{RMW}} &= \lambda y, x.\, G.\mathtt{val}_{\mathsf{W}}[G.\mathsf{W}^{\neq\max}_x \setminus dom(R ; [G.\mathsf{E}^{\sqsupseteq\mathtt{rel}}]; ([G.\mathsf{F}]; G.\mathsf{po})^{?}; G.\mathsf{rf}^{*}_{\overline{T}} ; [G.w^{\max}_y] \cup R_{\mathsf{RMW}})
\end{aligned}
$$

where $G.\mathsf{W}^{\neq\max}_x = G.\mathsf{W}_x \setminus \{G.w^{\max}_x\}$, $R \triangleq G.\mathsf{mo} ; G.\mathsf{rf}^{?}_{\overline{T}} ; G.\mathsf{hb}^{?}_{\overline{T}}$, and $R_{\mathsf{RMW}} \triangleq G.\mathsf{rf} ; [G.\mathsf{RMW}]$.

To explain these equations (which serve as our simulation invariants), we call a value v of location x *non-robustly readable* for thread τ if thread τ can read v from x under RC20 from some write event that is not the mo-maximal write to x in the current graph. Thus, a value v of x is *not* non-robustly readable for thread τ if τ is already aware of some mo-later write to x ("aware" here means an $\mathsf{rf}^{?}_{\overline{T}} ; \mathsf{hb}^{?}_{\overline{T}}$-path). Then, $V^{\mathsf{c}}(\tau, x)$ consists of all non-robustly readable values of x for thread τ; $V^{\mathsf{a}}(\tau, x)$ consists of all non-robustly readable values of x for thread τ that will remain non-robustly readable for τ even after it executes an acquire fence; and $V^{\mathsf{r}}(\tau, x)$ consists of all values of x that may remain non-robustly readable for any other thread even when it synchronizes (via the sw relation) with the last release fence of thread τ. In turn, $V^{\mathsf{w}}(y, x)$ consists of all values of x that may remain non-robustly readable for any thread after it reads (the most recent value) from $y \neq x$. The $V^{\alpha}_{\mathsf{RMW}}$ variants are similar, but they refer to non-robustly *writable* values for thread τ, that is: values of non-mo-maximal write events that thread τ may overwrite by putting the new write as their mo-successors.

Algorithm 1 Thread views maintenance for a step of thread τ labeled with $l \in \mathsf{Lab}$ with $\mathtt{typ}(l) \in \{\mathtt{R}, \mathtt{W}, \mathtt{RMW}, \mathtt{R}^\star\}$ and $\mathtt{loc}(l) = x$

```
1: $\forall \alpha \in \{\mathsf{a}, \mathsf{c}\} :\ V^{\alpha}(\tau, x) := \emptyset$
2: if $\mathtt{typ}(l) \in \{\mathtt{W}, \mathtt{RMW}\}$ then
3:   $V^{\mathsf{r}}(\tau, x) :\cup= \{M(x)\}$
4: if $\mathtt{typ}(l) \in \{\mathtt{R}, \mathtt{RMW}, \mathtt{R}^\star\}$ then
5:   $\forall y \neq x :\ V^{\mathsf{a}}(\tau, y) :\cap= V^{\mathsf{w}}(x, y)$
6:   if $\mathtt{mod}(l) \sqsupseteq \mathsf{acq}$ then
7:     $\forall y \neq x :\ V^{\mathsf{c}}(\tau, y) :\cap= V^{\mathsf{w}}(x, y)$
8: if $\mathtt{typ}(l) \in \{\mathtt{W}, \mathtt{RMW}\}$ then
9:   $\forall \alpha \in \{\mathsf{a}, \mathsf{c}, \mathsf{r}\}, \pi \neq \tau :\ V^{\alpha}(\pi, x) :\cup= \{M(x)\}$
```

Algorithm 2 Thread RMW-views maintenance for a step of thread τ labeled with $l \in \mathsf{Lab}$ with $\mathtt{typ}(l) \in \{\mathtt{R}, \mathtt{W}, \mathtt{RMW}, \mathtt{R}^\star\}$ and $\mathtt{loc}(l) = x$

```
1: $\forall \alpha \in \{\mathsf{a}, \mathsf{c}\} :\ V^{\alpha}_{\mathsf{RMW}}(\tau, x) := \emptyset$
2: if $\mathtt{typ}(l) = \mathtt{W}$ then
3:   $V^{\mathsf{r}}_{\mathsf{RMW}}(\tau, x) :\cup= \{M(x)\}$
4: if $\mathtt{typ}(l) \in \{\mathtt{R}, \mathtt{RMW}, \mathtt{R}^\star\}$ then
5:   $\forall y \neq x :\ V^{\mathsf{a}}_{\mathsf{RMW}}(\tau, y) :\cap= V^{\mathsf{w}}_{\mathsf{RMW}}(x, x)$
6:   if $\mathtt{mod}(l) \sqsupseteq \mathsf{acq}$ then
7:     $\forall y \neq x :\ V^{\mathsf{c}}_{\mathsf{RMW}}(\tau, y) :\cap= V^{\mathsf{w}}_{\mathsf{RMW}}(x, y)$
8: if $\mathtt{typ}(l) = \mathtt{W}$ then
9:   $\forall \alpha \in \{\mathsf{a}, \mathsf{c}, \mathsf{r}\}, \pi \neq \tau :\ V^{\alpha}_{\mathsf{RMW}}(\pi, x) :\cup= \{M(x)\}$
```

Algorithm 3 Thread views maintenance for a step of thread τ labeled with $\mathtt{F}(o_{\mathtt{F}})$

```
1: if $o_{\mathtt{F}} \sqsupseteq \mathsf{acq}$ then
2:     $\forall x :\ V^{\mathsf{c}}(\tau, x) := V^{\mathsf{a}}(\tau, x)$
3:     $\forall x :\ V^{\mathsf{c}}_{\mathsf{RMW}}(\tau, x) := V^{\mathsf{a}}_{\mathsf{RMW}}(\tau, x)$
4: if $o_{\mathtt{F}} \sqsupseteq \mathtt{rel}$ then
5:     $\forall x :\ V^{\mathsf{r}}(\tau, x) := V^{\mathsf{c}}(\tau, x)$
6:     $\forall x :\ V^{\mathsf{r}}_{\mathsf{RMW}}(\tau, x) := V^{\mathsf{c}}_{\mathsf{RMW}}(\tau, x)$
```

Algorithm 4
Location view maintenance for $l = \mathtt{W}(o, x, \_)$

```
1: $\forall y \neq x : V^{\mathsf{w}}(y, x) :\cup= \{M(x)\}$
2: $\forall y \neq x : V^{\mathsf{w}}_{\mathsf{RMW}}(y, x) :\cup= \{M(x)\}$
3: if $o \sqsupseteq \mathtt{rel}$ then
4:   $\forall y \neq x :\ V^{\mathsf{w}}(x, y) := V^{\mathsf{c}}(\tau, y)$
5:   $\forall y \neq x :\ V^{\mathsf{w}}_{\mathsf{RMW}}(x, y) := V^{\mathsf{c}}_{\mathsf{RMW}}(\tau, y)$
6: else
7:   $\forall y \neq x :\ V^{\mathsf{w}}(x, y) := V^{\mathsf{r}}(\tau, y)$
8:   $\forall y \neq x :\ V^{\mathsf{w}}_{\mathsf{RMW}}(x, y) := V^{\mathsf{r}}_{\mathsf{RMW}}(\tau, y)$
```

Algorithm 5
Location view maintenance for $l = \mathtt{RMW}(o, x, \_, \_)$

```
1: $\forall y \neq x : V^{\mathsf{w}}(y, x) :\cup= \{M(x)\}$
2:
3: if $o \sqsupseteq \mathtt{rel}$ then
4:   $\forall y \neq x :\ V^{\mathsf{w}}(x, y) :\cap= V^{\mathsf{c}}(\tau, y)$
5:   $\forall y \neq x :\ V^{\mathsf{w}}_{\mathsf{RMW}}(x, y) :\cap= V^{\mathsf{c}}_{\mathsf{RMW}}(\tau, y)$
6: else
7:   $\forall y \neq x :\ V^{\mathsf{w}}(x, y) :\cap= V^{\mathsf{r}}(\tau, y)$
8:   $\forall y \neq x :\ V^{\mathsf{w}}_{\mathsf{RMW}}(x, y) :\cap= V^{\mathsf{r}}_{\mathsf{RMW}}(\tau, y)$
```

We note that the following invariants always hold:

$$V^{\mathsf{a}}(\tau, x) \subseteq V^{\mathsf{c}}(\tau, x) \subseteq V^{\mathsf{r}}(\tau, x) \qquad V^{\mathsf{a}}_{\mathsf{RMW}}(\tau, x) \subseteq V^{\mathsf{c}}_{\mathsf{RMW}}(\tau, x) \subseteq V^{\mathsf{r}}_{\mathsf{RMW}}(\tau, x)$$
$$\forall \alpha \in \{\mathsf{a}, \mathsf{c}, \mathsf{r}\} : V^{\alpha}_{\mathsf{RMW}}(\tau, x) \subseteq V^{\alpha}(\tau, x) \qquad V^{\mathsf{w}}_{\mathsf{RMW}}(y, x) \subseteq V^{\mathsf{w}}(y, x)$$

Initially, since each location has only one write in the initial graph, we start with $V_{\emptyset}$ where all components always return the empty set of values. To maintain V, we define a 5-ary relation $\langle M, V \rangle \xrightarrow{\tau, l} V'$ which ascribes how to update V when thread τ takes a step labeled with l starting from memory M. We define this relation using "pseudo-code update algorithms". For every transition, we first "execute" Algorithm 1 and Algorithm 2 to update V^{α} and $V^{\alpha}_{\mathsf{RMW}}$ (for $\alpha \in \{\mathsf{a}, \mathsf{c}, \mathsf{r}\}$) or Algorithm 3 for fence steps. Then, V^{w} and $V^{\mathsf{w}}_{\mathsf{RMW}}$ are updated using Algorithm 4 for write steps or Algorithm 5 for RMW steps. In these algorithms we write $X :\cup= Y$ for $X := X \cup Y$ and $X :\cap= Y$ for $X := X \cap Y$.

Algorithm 1 updates V^{c}, V^{a}, and V^{r}. To understand its steps, consider an access to location x by thread τ. Since we are running under SC, every such access will make τ aware of the most recent write to x, so that there are not any non-robustly readable values of x for thread τ. Accordingly, Line 1 makes $V^{\mathsf{c}}(\tau, x)$ and $V^{\mathsf{a}}(\tau, x)$ empty. If τ writes to x, then the previous value of x in the memory may be non-robustly readable for every other thread that will synchronize with the last

SILENT
$$\frac{\begin{array}{c} L = \langle \varepsilon, R_{\mathrm{in}}, R_{\mathrm{out}} \rangle \\ R_{\mathrm{in}} \cap R_{\mathrm{taint}} = \emptyset \implies R'_{\mathrm{taint}} = R_{\mathrm{taint}} \setminus R_{\mathrm{out}} \\ R_{\mathrm{in}} \cap R_{\mathrm{taint}} \neq \emptyset \implies R'_{\mathrm{taint}} = R_{\mathrm{taint}} \cup R_{\mathrm{out}} \end{array}}{\langle M, R_{\mathrm{taint}}, V_{\mathsf{SC}}, V \rangle \xrightarrow{\tau,L}_{\mathsf{SC_M}} \langle M, R'_{\mathrm{taint}}, V_{\mathsf{SC}}, V \rangle}$$

NON-READ
$$\frac{\begin{array}{c} \mathtt{typ}(L) \in \{\mathtt{W}, \mathtt{RMW}, \mathtt{R}^{\star}\} \qquad x = \mathtt{loc}(L) \\ \mathtt{typ}(L) \in \{\mathtt{RMW}, \mathtt{R}^{\star}\} \implies \mathtt{val}_{\mathtt{R}}(L) = M(x) \\ \mathtt{typ}(L) \in \{\mathtt{W}, \mathtt{RMW}\} \implies M' = M[x \mapsto \mathtt{val}_{\mathtt{W}}(L)] \\ \mathtt{typ}(L) = \mathtt{R}^{\star} \implies M' = M \\ R'_{\mathrm{taint}} = R_{\mathrm{taint}} \setminus \mathsf{R}_{\mathsf{out}}(L) \\ V_{\mathsf{SC}} \xrightarrow{\tau,\mathrm{lab}(L)} V'_{\mathsf{SC}} \qquad \langle M, V \rangle \xrightarrow{\tau,\mathrm{lab}(L)} V' \end{array}}{\langle M, R_{\mathrm{taint}}, V_{\mathsf{SC}}, V \rangle \xrightarrow{\tau,L}_{\mathsf{SC_M}} \langle M', R'_{\mathrm{taint}}, V'_{\mathsf{SC}}, V' \rangle}$$

FENCE
$$\frac{\mathtt{typ}(L) = \mathtt{F} \qquad \langle M, V \rangle \xrightarrow{\tau,\mathrm{lab}(L)} V'}{\langle M, R_{\mathrm{taint}}, V_{\mathsf{SC}}, V \rangle \xrightarrow{\tau,L}_{\mathsf{SC_M}} \langle M, R_{\mathrm{taint}}, V_{\mathsf{SC}}, V' \rangle}$$

CONTROL
$$\frac{\mathtt{typ}(L) = \mathtt{CTRL}}{\langle M, R_{\mathrm{taint}}, V_{\mathsf{SC}}, V \rangle \xrightarrow{\tau,L}_{\mathsf{SC_M}} \langle M, R_{\mathrm{taint}}, V_{\mathsf{SC}}, V \rangle}$$

READ
$$\frac{\begin{array}{c} \mathtt{typ}(L) = \mathtt{R} \qquad x = \mathtt{loc}(L) \qquad \mathtt{val}_{\mathtt{R}}(L) = M(x) \\ x \notin V_{\mathsf{SC}}(\tau) \lor V^{\mathsf{c}}(\tau, x) = \emptyset \qquad R'_{\mathrm{taint}} = R_{\mathrm{taint}} \setminus \mathsf{R}_{\mathsf{out}}(L) \\ V_{\mathsf{SC}} \xrightarrow{\tau,\mathrm{lab}(L)} V'_{\mathsf{SC}} \qquad \langle M, V \rangle \xrightarrow{\tau,\mathrm{lab}(L)} V' \end{array}}{\langle M, R_{\mathrm{taint}}, V_{\mathsf{SC}}, V \rangle \xrightarrow{\tau,L}_{\mathsf{SC_M}} \langle M, R'_{\mathrm{taint}}, V'_{\mathsf{SC}}, V' \rangle}$$

TAINT-READ
$$\frac{\begin{array}{c} \mathtt{typ}(L) = \mathtt{R} \qquad x = \mathtt{loc}(L) \\ x \in V_{\mathsf{SC}}(\tau) \qquad V^{\mathsf{c}}(\tau, x) \neq \emptyset \\ R'_{\mathrm{taint}} = R_{\mathrm{taint}} \cup \mathsf{R}_{\mathsf{out}}(L) \qquad V_{\mathsf{SC}} \xrightarrow{\tau,\mathrm{lab}(L)} V'_{\mathsf{SC}} \end{array}}{\langle M, R_{\mathrm{taint}}, V_{\mathsf{SC}}, V \rangle \xrightarrow{\tau,L}_{\mathsf{SC_M}} \langle M, R'_{\mathrm{taint}}, V'_{\mathsf{SC}}, V \rangle}$$

ERR-SC
$$\frac{\begin{array}{c} \mathtt{typ}(L) \in \{\mathtt{W}, \mathtt{RMW}, \mathtt{R}^{\star}\} \qquad x = \mathtt{loc}(L) \qquad x \in V_{\mathsf{SC}}(\tau) \\ \mathtt{typ}(L) = \mathtt{W} \implies V^{\mathsf{c}}_{\mathtt{RMW}}(\tau, x) \neq \emptyset \\ \mathtt{typ}(L) = \mathtt{RMW} \implies \mathtt{val}_{\mathtt{R}}(L) \in V^{\mathsf{c}}_{\mathtt{RMW}}(\tau, x) \\ \mathtt{typ}(L) = \mathtt{R}^{\star} \implies \mathtt{val}_{\mathtt{R}}(L) \in V^{\mathsf{c}}(\tau, x) \end{array}}{\langle M, R_{\mathrm{taint}}, V_{\mathsf{SC}}, V \rangle \xrightarrow{\tau,L}_{\mathsf{SC_M}} \bot}$$

ERR-DEP
$$\frac{\begin{array}{c} \mathtt{typ}(L) \in \{\mathtt{W}, \mathtt{RMW}, \mathtt{R}^{\star}, \mathtt{CTRL}\} \\ \mathsf{R}_{\mathsf{in}}(L) \cap R_{\mathrm{taint}} \neq \emptyset \end{array}}{\langle M, R_{\mathrm{taint}}, V_{\mathsf{SC}}, V \rangle \xrightarrow{\tau,L}_{\mathsf{SC_M}} \bot}$$

Fig. 5. $\mathsf{SC_M}$ transitions.

release fence of thread τ (Line 3). For other locations, if τ reads from x it is now confined as the write event view of x ascribes (Lines 5 and 7). A relaxed read will only affect other locations once an acquire fence is placed (only $V^{\mathsf{a}}(\tau, y)$ is restricted). Finally, if τ writes to x, then the previous (overwritten) value of x in the memory is non-robustly readable by all other threads (Line 9).

Algorithm 2 updates $V^{\mathsf{c}}_{\mathtt{RMW}}$, $V^{\mathsf{a}}_{\mathtt{RMW}}$, and $V^{\mathsf{r}}_{\mathtt{RMW}}$. Its steps are exactly the same as Algorithm 1, where the only exceptions are in Line 3 and Line 9, which should not be preformed if the access of thread τ is an RMW. In this case, the event writing the current value $M(x)$ will be read by the RMW event added to graph in thread τ, and so, to satisfy ATOMICITY, it should not affect the values that are non-robustly writable by other threads.

Algorithm 3 updates the thread views when fences are performed. For an acquire fence, this requires to propagate the acquire views into the current views (Lines 2 and 3); and for a release fence, we propagate the current views into the release views (Lines 5 and 6).

Algorithm 4 updates the location views when non-RMW writes are performed. Lines 1 and 2 reflect the fact that after writing to x by thread τ, the previous value of x in the memory may be non-robustly readable/writable for every other thread that will read from some other location y. In addition, after reading from x, other threads will be confined by what thread τ "releases" in its write: the current view for release writes or the release view for relaxed writes. Algorithm 5 is similar. To support release sequences, Lines 4, 5, 7 and 8, "absorb" the thread view in the location view instead of overwriting it.

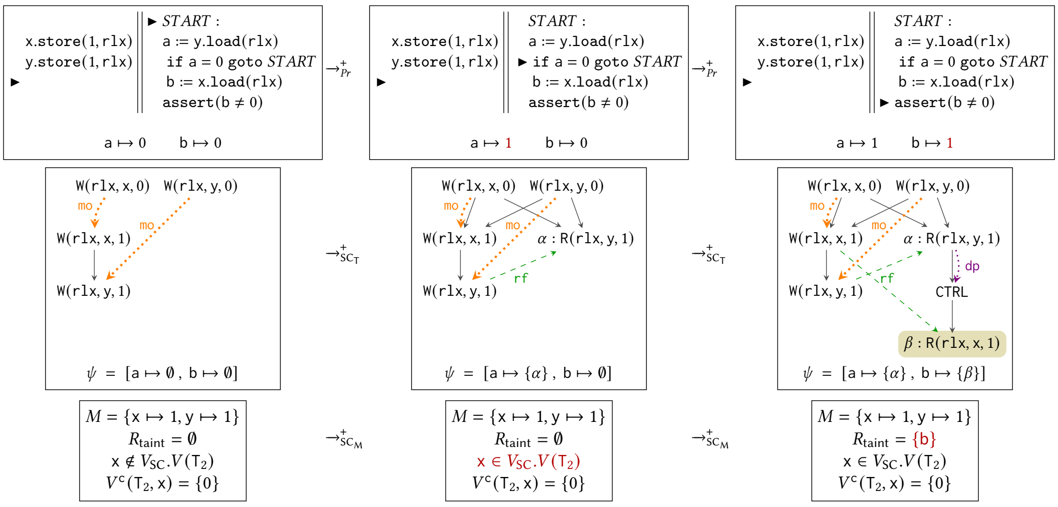
Using these algorithms, we can define the (finite) memory system $\mathrm{SC_M}$.

Definition 5.3. The memory system $\mathrm{SC_M}$ is given by:

- Its states consists of the error state $\bot$, and all tuples of the form $\langle M, R_{\text{taint}}, V_{\mathrm{SC}}, V\rangle$ (where each of these components is of the form described above).
- Its initial state is $\langle M_\emptyset, R_{\text{taint}\emptyset}, V_{\mathrm{SC}\emptyset}, V_\emptyset\rangle$ (all as defined above).
- Its transitions are given in Fig. 5.

The transitions of $\mathrm{SC_M}$ in Fig. 5 follow the descriptions above, using the predefined $V_{\mathrm{SC}} \longrightarrow V'_{\mathrm{SC}}$ and $\langle M, V\rangle \longrightarrow V'$ relations. In particular, a register is marked as taint in a read step reading from x by thread τ iff some value of x may be non-robustly read by thread τ ($V^{\mathrm{c}}(\tau, x) \neq \emptyset$) while the maximal write to x must be executed before τ's read step in SC-executions ($x \in V_{\mathrm{SC}}(\tau)$). The ERR-SC and ERR-DEP steps detect robustness violations: either by performing a non-robust non-read transition, or by depending on a taint read.

Example 5.4. We demonstrate how $\mathrm{SC_M}$ simulates $\mathrm{SC_T}$ for establishing the non-observational-robustness of the (non-fenced) MP program in Ex. 4.2 by reaching the witness discussed in Ex. 4.15. The following illustration depicts three program states (top part) during a run, the corresponding execution graphs in the run of $\mathrm{SC_T}$ (middle part; the set T consists of the highlighted events), and the corresponding states in the run of $\mathrm{SC_M}$ (bottom part). To simplify the presentation, we only present the part of $\mathrm{SC_M}$'s states that is relevant for detecting the robustness violation (which is relatively small in this example).


x.store(1, rlx)
y.store(1, rlx)
START :
a := y.load(rlx)
if a = 0 goto START
b := x.load(rlx)
assert(b ≠ 0)
a ↦ 0   b ↦ 0
a ↦ 1   b ↦ 0
a ↦ 1   b ↦ 1
$\xrightarrow{+}_{Pr}$
W(rlx, x, 0)   W(rlx, y, 0)
W(rlx, x, 1)
W(rlx, y, 1)
α : R(rlx, y, 1)
CTRL
β : R(rlx, x, 1)
mo
rf
dp
ψ = [a ↦ ∅ , b ↦ ∅]
ψ = [a ↦ {α} , b ↦ ∅]
ψ = [a ↦ {α} , b ↦ {β}]
$\xrightarrow{+}_{\mathrm{SC_T}}$
$M = \{x \mapsto 1, y \mapsto 1\}$
$R_{\text{taint}} = \emptyset$
$x \notin V_{\mathrm{SC}}.V(\mathrm{T_2})$
$V^{\mathrm{c}}(\mathrm{T_2}, x) = \{0\}$
$x \in V_{\mathrm{SC}}.V(\mathrm{T_2})$
$R_{\text{taint}} = \{b\}$
$\xrightarrow{+}_{\mathrm{SC_M}}$

To reach the first (leftmost) state, we run the first thread until it completes its execution. At this point we have $x = y = 1$ in the memory, no tainted registers ($R_{\text{taint}} = \emptyset$), the second thread is not "$\mathrm{hb_{SC}}$-aware" of the maximal write to x ($\mathrm{x} \notin V_{\mathrm{SC}}.V(\mathrm{T_2})$), and the second thread may non-robustly read (i.e., can read under RC20 but not under SC if we continue from this state) the value 0 from x ($V^{\mathrm{c}}(\mathrm{T_2}, \mathrm{x}) = \{0\}$). Then, since we are running under (instrumented) SC, the second thread has to read 1 from y. This does not change the memory or the set of tainted registers, but it does make the thread "$\mathrm{hb_{SC}}$-aware" of the maximal write to x (via the added rf-edge on y). Hence, we have $\mathrm{x} \in V_{\mathrm{SC}}.V(\mathrm{T_2})$ in the second state. Next, when the second thread reads x, it marks b as a tainted register—the thread is "$\mathrm{hb_{SC}}$-aware" of the maximal write to x but can non-robustly read some value of x ($V^{\mathrm{c}}(\mathrm{T_2}, \mathrm{x}) = \{0\}$). Finally, when we use the value of the tainted register b (by executing the assertion) a robustness violation is detected via $\mathrm{SC_M}$'s ERR-DEP transition.

If we include a release fence before $\mathtt{y.store}(1, \mathtt{rlx})$ in the first thread and an acquire fence before $\mathtt{b} := \mathtt{x.load}(\mathtt{rlx})$ in the second thread, the register $\mathtt{b}$ will not become tainted, and the program will be (correctly) deemed as robust:

- After the first thread executes the release fence, we will have $V^{\mathsf{r}}(\mathtt{T}_1, \mathtt{x}) = \emptyset$;
- Then, after the first thread executes $\mathtt{y.store}(1, \mathtt{rlx})$, we will have $V^{\mathsf{w}}(\mathtt{y}, \mathtt{x}) = \emptyset$;
- Then, after the second thread reads 1 from $\mathtt{y}$, we will have $V^{\mathsf{a}}(\mathtt{T}_2, \mathtt{x}) = \{0\} \cap \emptyset = \emptyset$;
- Then, after the second thread executes the acquire fence, we will have $V^{\mathsf{c}}(\mathtt{T}_2, \mathtt{x}) = \emptyset$;
- Then, when the second thread reads $\mathtt{x}$, it cannot non-robustly read any value of $\mathtt{x}$, so the register $\mathtt{b}$ remains untainted.

Next, we establish the equivalence of SC_T and its finite abstraction SC_M, and derive our final results as simple corollaries of Theorems 4.16, 5.2 and 5.5 and Corollary 5.6.

Theorem 5.5. *A state $\langle \overline{q}, \bot \rangle$ is reachable in $Pr \parallel \mathsf{SC}_\mathsf{T}$ iff it is reachable in $Pr \parallel \mathsf{SC}_\mathsf{M}$.*

Corollary 5.6. *There exists a non-observational-robustness witness for Pr iff $\langle \overline{q}, \bot \rangle$ is reachable in $Pr \parallel \mathsf{SC}_\mathsf{M}$ for some $\overline{q} \in Pr.\mathsf{Q}$,*

Corollary 5.7. *If $\langle \overline{q}, \bot \rangle$ is not reachable in $Pr \parallel \mathsf{SC}_\mathsf{M}$ for every $\overline{q} \in Pr.\mathsf{Q}$, then Pr is observationally robust.*

Finally, a decision procedure for robustness follows from Thm. 4.6, Lemma 4.20, and Corollary 5.6.

Corollary 5.8. *A program Pr is robust iff $\langle \overline{q}, \bot \rangle$ is not reachable in $Pr' \parallel \mathsf{SC}_\mathsf{M}$ for every $\overline{q} \in Pr'.\mathsf{Q}$, where Pr' is the program obtained from Pr by adding a (vacuous) assert instruction $\mathtt{assert}(r = r)$ after every read instruction $r := x.\mathtt{load}(o_\mathsf{R})$.*

It also follows that robustness verification against RC20 is PSPACE-complete. The upper bound follows by reduction to reachability in an instrumented SC semantics; whereas the lower bound directly follows from [Lahav and Margalit 2019], as the model studied there is a fragment of RC20.

6 EXTENSION WITH NON-ATOMICS

The extension of our results to cover C11-style non-atomics with "catch-fire" semantics is straightforward: we can identify data-races on non-atomics while checking for robustness. Next, we describe this extension. We refer to the extended model as RC20+NA.

First, we include "na" in the set Mod of access modes, where na is included both in Mod_R (read access modes) and in Mod_W (write access modes). We assume a set Loc_na of *non-atomic locations* disjoint from the set Loc of *atomic locations*, and require that all accesses to the locations in Loc_na are na-accesses, and conversely, that all na-accesses are to locations in Loc_na. We further assume that non-atomic accesses are used in plain loads and stores, but not in RMWs or wait instructions.

To give semantics to programs with non-atomic accesses, we define *data race* on non-atomics, and consider consistent execution graphs that have such races as program failures.

Definition 6.1. Two events a and b are called *conflicting* in an execution graph G if $a, b \in G.\mathsf{E}$, $G.\mathtt{loc}(a) = G.\mathtt{loc}(b)$, and $\mathtt{W} \in \{G.\mathtt{typ}(a), G.\mathtt{typ}(b)\}$. A pair $\langle a, b \rangle$ is called a *race* in G if a and b are conflicting events in G and $\langle a, b \rangle \notin G.\mathsf{hb} \cup G.\mathsf{hb}^{-1}$. An execution graph G is called *racy* if there is some race $\langle a, b \rangle$ in G with $\mathsf{na} \in \{G.\mathtt{mod}(a), G.\mathtt{mod}(b)\}$.

Definition 6.2. A program *Pr may fail under* RC20+NA if either (i) it may fail under RC20 (Def. 3.12) or (ii) some RC20-consistent execution graph G generated by Pr is racy.

Our goal is to extend SC_T, and, in turn, SC_M, so that these systems can be used in order to establish the fact that a given program may fail under RC20+NA iff it may fail under SC. We note that the definition of failure under SC is unchanged: for the SC memory system non-atomic accesses are not at all distinguished from atomic ones. To do so, we apply a simple program transformation:

(1) We split every $x \in \mathsf{Loc}_{\mathsf{na}}$ into $|\mathsf{Tid}|$ variables, $x_1, \dots, x_{|\mathsf{Tid}|}$, one for each thread.
(2) We replace every (non-atomic) write to x with a sequence of $|\mathsf{Tid}|$ (non-atomic) writes writing to $x_1, \dots, x_{|\mathsf{Tid}|}$ the value that was written to x.
(3) We replace every (non-atomic) read operation from x of thread T_i with a read from x_i followed by a write (of the read value that was read) to x_i.

We denote by $\mathsf{trans}(Pr)$ the program obtained by applying this transformation on Pr.

Example 6.3. Applying the above transformation to the program on the left generates the program on the right:

$$\begin{array}{l}\mathtt{a := x.load(na)}\\ \mathtt{y.store(1,rlx)}\end{array}\;\Bigg\|\;\begin{array}{l}\mathtt{wait(y = 1,rlx)}\\ \mathtt{b := x.load(na)}\\ \mathtt{y.store(2,rel)}\end{array}\;\Bigg\|\;\begin{array}{l}\mathtt{wait(y = 2,acq)}\\ \mathtt{x.store(1,na)}\end{array}\quad\rightsquigarrow\quad\begin{array}{l}\mathtt{a := x_1.load(na)}\\ \mathtt{x_1.store(a,na)}\\ \mathtt{y.store(1,rlx)}\end{array}\;\Bigg\|\;\begin{array}{l}\mathtt{wait(y = 1,rlx)}\\ \mathtt{b := x_2.load(na)}\\ \mathtt{x_2.store(b,na)}\\ \mathtt{y.store(2,rel)}\end{array}\;\Bigg\|\;\begin{array}{l}\mathtt{wait(y = 2,acq)}\\ \mathtt{x_1.store(1,na)}\\ \mathtt{x_2.store(1,na)}\\ \mathtt{x_3.store(1,na)}\end{array}$$

This program transformation essentially ensures that all races are expressed as write-write races. We use per-thread variables to make sure that two concurrent reads to the same location in the original program will not induce a race in the transformed program. Applying this transformation allows us to detect races by checking that whenever a thread executes a non-atomic access it is already aware of the latest write to the same (non-atomic) location. Since our instrumented semantics is designed to track the set of locations of which every is aware to the latest write, this approach requires only minimal changes in SC_T and SC_M. Formally, we have the following:

Lemma 6.4. *The following are equivalent for every concurrent program Pr:*

(*i*) *Some* RC20*-consistent graph generated by Pr is racy.*
(*ii*) *There exists some* RC20*-consistent execution graph G that is generated by* $\mathsf{trans}(Pr)$ *and a* $(G.\mathsf{po} \cup G.\mathsf{rf})$*-maximal event* $e \in G.\mathsf{E}$ *such that* $G.\mathsf{mod}(e) = \mathsf{na}$ *and* $G'.w^{\max}_{G'.\mathtt{loc}(e)} \notin dom(G'.\mathsf{rf}^? \,;\, G'.\mathsf{hb} \,;\, [\mathsf{E}^{\mathtt{tid}(e)}])$ *for* $G' = G \setminus \{e\}$.

Condition (*ii*) above is similar to the condition that SC_T (and, in turn, SC_M) monitors, so that by using the transformation above we do not need to change the instrumentation in SC_T (and SC_M). Indeed, it suffices to add the following transition to SC_T (we refer to the extend system as $\mathsf{SC}^{\mathsf{na}}_\mathsf{T}$):

$$\frac{\mathtt{mod}(L) = \mathsf{na} \qquad G.w^{\max}_{\mathtt{loc}(L)} \notin dom(G.\mathsf{rf}^?_{\overline{T}} \,;\, G.\mathsf{hb}_{\overline{T}} \,;\, [\mathsf{E}^\tau])}{\langle G, \psi, T\rangle \xrightarrow{\tau,L}_{\mathsf{SC}^{\mathsf{na}}_\mathsf{T}} \bot}$$

This additional transition allows the system to move to $\bot$ when a race is detected.

Theorem 6.5. *If* $\langle \overline{q}, \bot\rangle$ *is not reachable in* $\mathsf{trans}(Pr) \parallel \mathsf{SC}^{\mathsf{na}}_\mathsf{T}$ *for every* $\overline{q} \in \mathsf{trans}(Pr).\mathsf{Q}$*, then Pr may fail under* RC20+NA *iff Pr may fail under* SC.

Finally, we extend SC_M with the following step that precisely matches the step added to SC_T (we refer to the extend system as $\mathsf{SC}^{\mathsf{na}}_\mathsf{M}$):

$$\frac{\mathtt{mod}(L) = \mathsf{na} \qquad V^{\mathsf{c}}(\tau, \mathtt{loc}(L)) \neq \emptyset}{\langle M, R_{\mathsf{taint}}, V_{\mathsf{SC}}, V\rangle \xrightarrow{\tau,L}_{\mathsf{SC}^{\mathsf{na}}_\mathsf{M}} \bot}$$

The simulation argument relating SC_T and SC_M is easily extended to relate $\mathsf{SC}^{\mathsf{na}}_\mathsf{T}$ and $\mathsf{SC}^{\mathsf{na}}_\mathsf{M}$, which provides us with our final result:

Corollary 6.6. *If* $\langle \overline{q}, \bot\rangle$ *is not reachable in* $\mathsf{trans}(Pr) \parallel \mathsf{SC}^{\mathsf{na}}_\mathsf{M}$ *for every* $\overline{q} \in \mathsf{trans}(Pr).\mathsf{Q}$*, then Pr may fail under* RC20+NA *iff Pr may fail under* SC.

7 IMPLEMENTATION AND EVALUATION

We have implemented our algorithm as an extension of "Rocker" [Lahav and Margalit 2019]—a robustness checking tool against the RA model (a fragment of RC20).[5] The implementation transforms an input program in a minimal C-like language to an instrumented Promela program that can be verified by the SPIN model checker [Holzmann 1997]. The instrumentation follows the description above, but instead of checking reachability in an instrumented SC semantics, we instrument the program itself, and equivalently check for reachability of a state representing an observational robustness violation under the usual SC interleaving semantics.

Remark 3. Similarly to what was done in [Lahav and Margalit 2019], as an implementation optimization, it is possible to group together different values, making the range of the different views in the instrumented semantics to be the powerset of some fixed partition of Val rather than the powerset of Val (which corresponds to the powerset of the trivial partition $\{\{v\} \mid v \in \mathsf{Val}\}$). The selected partition should be sufficiently expressive for checking whether enabled successful CAS/BCAS/wait instructions may read a stale value. In our implementation, for every location x, we use a partition of the form $\{\{v\} \mid v \in \mathsf{Crit}_x\} \cup \{\mathsf{Val} \setminus \mathsf{Crit}_x\}$, where Crit_x are all values that successful CAS/BCAS/wait instructions of location x may read. (We require the user to annotate the input program with this information.) In particular, for programs that have no CAS/BCAS/wait instructions at all, all views are boolean predicates (rather than functions to $\mathcal{P}(\mathsf{Val})$).

We have performed a series of experiments on litmus tests, examples from [Lahav and Margalit 2019], and additional algorithms that have been shown to be challenging-to-get-right. Figure 6 summarizes our evaluation results when run on an Intel®Core™i5-6300U CPU @2.40GHz GNU/Linux machine. In several cases, for verifying robustness, we need to replace busy-loops with wait instructions or use BCAS instead of CAS. Clearly, such transformations preserve the program semantics. The columns "Program" and "#T" show the benchmark name and its number of threads. The column "Rob" indicates whether the program is robust ("R") and observationally robust ("O"). The "Time" column presents the time that was needed to automatically verify robustness for each method (i.e., detecting non-robustness witnesses vs. detecting non-observational-robustness witnesses), as well as the running time of SPIN on the given program without any instrumentation (in the column named "SC"). The latter may shed light on the size of the each verification problem and the overhead required for robustness verification w.r.t. standard explicit model checking. The number in parentheses indicates the percentage of the time used for compiling the verifier produced by SPIN (using gcc -O2), which, in some cases, dominates the overall search time. We note that the time needed for Rocker to create the instrumented program and for SPIN to create the verifier are negligible (< 0.3s).

We interpret these results as showing that the time difference between merely checking robustness vs. observational robustness is rather small. Naturally, for non-robust but observationally robust programs, we have a significant gap.

For comparison with existing work, we have run our tool on programs that employ only release/acquire accesses from [Lahav and Margalit 2019]. Due to the more fine-grained instrumentation required to handle the full RC20 model, we obtain slightly increased runtime compared to the RA-only robustness verifier. This can be seen in Fig. 7 (all programs there are robust against RA).

Next, we describe some of the verified examples and our findings:

arc: An implementation of Rust's *Atomic Reference Counter* [Doko and Vafeiadis 2017; Rust 2020] used by three threads repeatedly going online and offline. It relies on the synchronization induced by relaxed RMWs inside release sequences, and our tool verified the robustness of this mechanism.

[5]Our implementation and the examples it was tested on are available in the artifact accompanying this paper.

Program	#T	Rob		Time (sec)		
		R	O	R	O	SC
arc	3	✓	✓	67.1 (8%)	71.2 (8%)	3.9 (26%)
peterson	2	✗	✗	1.1 (100%)	1.1 (100%)	0.9 (100%)
peterson-fix1	2	✓	✓	1.1 (100%)	1.1 (100%)	0.9 (100%)
peterson-fix2	2	✓	✓	1.1 (100%)	1.1 (100%)	0.9 (100%)
singleton	4	✓	✓	2.8 (96%)	3.0 (97%)	1.1 (100%)
singleton-rlx	4	✓	✓	2.8 (96%)	3.1 (97%)	1.1 (100%)
wait-free-ring	2	✓	✓	0.9 (100%)	1.0 (100%)	0.9 (100%)
dekker-rlx	2	✓	✓	1.5 (100%)	1.6 (100%)	0.9 (100%)
seqlock rdmw	3	✗	✓	7.1 (100%)	59.7 (12%)	30.9 (4%)
seqlock fence	3	✗	✓	8.0 (100%)	81.8 (10%)	39.9 (3%)
seqlock rdmw-rw	3	✗	✓	6.4 (100%)	53.5 (13%)	31.9 (3%)
seqlock fence-rw	3	✗	✓	6.7 (100%)	74.6 (9%)	41.5 (3%)
chase-lev	3	✓	✓	57.7 (2%)	58.8 (3%)	27.1 (4%)
lock exchange	2	✓	✓	1.0 (100%)	1.0 (100%)	0.9 (100%)
spinlock4-rlx	4	✓	✓	2.5 (52%)	2.9 (48%)	1.3 (77%)

Fig. 6. Evaluation results

Program	#T	Time (sec)		
		R	O	Rocker
peterson-ra	2	1.1 (100%)	1.1 (100%)	1.0 (100%)
dekker-ra	2	1.4 (100%)	1.4 (100%)	1.1 (100%)
seqlock-ra	3	62.2 (10%)	59.1 (10%)	58.3 (8%)
chase-lev-ra	3	59.8 (3%)	55.7 (3%)	39.9 (3%)
spinlock4-ra	4	2.4 (63%)	2.5 (60%)	2.3 (61%)
lamport2-3	3	102.3 (9%)	95.5 (10%)	69.9 (8%)
rcu	4	54.7 (5%)	46.1 (6%)	52.0 (4%)
rcu-offline	3	80.3 (13%)	77.5 (14%)	61.4 (12%)
ticketlock4	4	12.1 (12%)	11.8 (14%)	11.7 (13%)

Fig. 7. Comparison with *Rocker* [Lahav and Margalit 2019] on robust RA programs

peterson: An implementation of Peterson's lock for C11 by V'jukov [2008]. We (automatically) identified a robustness violation when one of the threads repeatedly tries to enter the critical section. Previous verification efforts (including an informal argument for correctness [Williams 2008], testing [V'jukov 2013], as well as a formal proof in a dedicated program logic [Dalvandi et al. 2020; Doherty et al. 2019]) focused on just a single attempt to enter the critical section by each thread, and, thus, did not detect this issue. We also note that the fact that both threads cannot simultaneously be in the critical section (which previous work established for a single critical section by each thread) does not suffice in C11: we need locked regions to be ordered by hb. We propose (and we verified) two fixes to the implementation: [fix1] Promoting a certain relaxed read to acquire; and [fix2] Placing an additional acquire fence before entering the critical section (and then we can demote an existing acquire read to be relaxed).

singleton: A double-checked locking pattern from the "boost" library [Bahmann and Blechmann 2012]. We found that robustness (and, consequently, correctness) still hold even if we demote the read inside the lock to be relaxed (thus suggesting a possible performance improvement).

wait-free-ring: A wait-free ring buffer with a single producer and a single consumer taken from the "boost" library [Bahmann and Blechmann 2012].

dekker: An optimized (rather tricky) implementation by Williams [2010] of the classical Dekker's mutual exclusion algorithm that employs relaxed accesses and SC fences.

seqlocks: Optimized implementations of Seqlock (see §4.1) based on [Boehm 2012] used by three threads (which non-deterministically write and read from the protected data structure). These programs are observationally robust, but not robust. We verified the two implementations of Boehm [2012]: one using "read-don't-modify-write" instructions ("seqlock rdmw"), and one using acquire fences ("seqlock fence"). We also tried a writer that uses efficient relaxed writes with appropriate release fences (no "rw" suffix), as well as a less efficient writer using release writes ("rw" suffix).

chase-lev: An optimized work stealing double-ended queue that uses relaxed accesses [Chase and Lev 2005; Kang 2018]. The queue owner inserts and removes from the bottom of the queue. Stealers steal from the top of the queue. In this case, to prove robustness, we had to make the write to "bot" (the bottom index) in the owner pop function to be release. We also had to add SC-fences in the owner push function between the relaxed read of "bot" and the read acquire of "top", as well as before the stealer starts stealing (to allow stealing loops). In this case, these strengthenings are due to limitations of the robustness correctness criterion, rather than bugs in the implementation.

8 RELATED WORK AND CONCLUSIONS

We provided a sound and precise robustness verification method against RC20, a slight variant of the RC11 memory model, which includes release/acquire and relaxed reads, writes, and RMWs, as well as release/acquire fences. To support speculative relaxed reads as used in seqlock algorithms, we introduced an observational robustness notion, and showed how it can be soundly verified. We implemented our method and evaluated it on multiple examples, demonstrating the effectiveness of this approach for detecting subtle bugs during the development of concurrent algorithms (or migrating algorithms written for SC) for a rather complex weak memory model.

Previous work studied robustness against hardware models, e.g., [Alglave et al. 2017; Alglave and Maranget 2011; Burckhardt et al. 2007; Derevenetc and Meyer 2014], and mostly against the x86-TSO model, e.g., [Abdulla et al. 2015b,a; Bouajjani et al. 2013, 2018, 2011; Burckhardt and Musuvathi 2008; Burnim et al. 2011; Gotsman et al. 2012; Linden and Wolper 2011, 2013; Liu et al. 2012; Owens 2010]. From a complexity perspective, our results show that verification of robustness against RC20 is similar to verification of robustness against TSO—they are both PSPACE-complete. Nevertheless, compared to TSO, the technical challenge here stems from the lack of (traditional) operational semantics, delicate synchronization definition, and rather weak RMW operations.

A (very imprecise) robustness criteria against a *programming language* memory model emerges from the well-known DRF guarantee [Adve and Hill 1990; Gharachorloo et al. 1992], and our "non-robustness witnesses" can be viewed as a precise DRF condition. More recently, as mentioned above, Lahav and Margalit [2019] provided a precise robustness analysis for release/acquire semantics. Our work builds and extends this result to include relaxed accesses (with $\mathrm{acyclic}(\mathsf{po} \cup \mathsf{rf})$) and release/acquire fences. Observational robustness is, to the best of our knowledge, a novel notion, arising when analyzing the use case of relaxed accesses as in the seqlock algorithm.

For shared memory distributed systems, robustness of transactional programs against serializability was studied in [Beillahi et al. 2019a,b; Bernardi and Gotsman 2016; Brutschy et al. 2018; Fekete et al. 2005; Nagar and Jagannathan 2018]. Except for [Beillahi et al. 2019a,b], these papers do not provide precise verification methods, but rather practical over-approximations.

ACKNOWLEDGMENTS

We thank the POPL'21 reviewers for their helpful feedback. This research was supported by the Israel Science Foundation (grant number 5166651). The second author was also supported by the Alon Young Faculty Fellowship.

REFERENCES

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

Parosh Aziz Abdulla, Mohamed Faouzi Atig, Magnus Lång, and Tuan Phong Ngo. 2015b. Precise and sound automatic fence insertion procedure under PSO. In *NETYS*. Springer International Publishing, Cham, 32–47.

Parosh Aziz Abdulla, Mohamed Faouzi Atig, and Tuan-Phong Ngo. 2015a. 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

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

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

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

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

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

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

Helge Bahmann and Tim Blechmann. 2012. Atomic usage examples. Retrieved june 08, 2020 from https://www.boost.org/doc/libs/1_55_0/doc/html/atomic/usage_examples.html

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

Sidi Mohamed Beillahi, Ahmed Bouajjani, and Constantin Enea. 2019a. Checking robustness against snapshot isolation. In *Computer Aided Verification*. Springer International Publishing, Cham, 286–304.

Sidi Mohamed Beillahi, Ahmed Bouajjani, and Constantin Enea. 2019b. Robustness against transactional causal consistency. In *CONCUR 2019*, Vol. 140. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 30:1–30:18. https://doi.org/10.4230/LIPIcs.CONCUR.2019.30

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

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

Hans-J. Boehm and Brian Demsky. 2014. Outlawing ghosts: avoiding out-of-thin-air results. In *MSPC*. ACM, New York, NY, USA, Article 7, 6 pages. https://doi.org/10.1145/2618128.2618134

Hans-J Boehm, Olivier Giroux, and Viktor Vafeiades. 2018. P0982R1: Weaken release sequences. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0982r1.html. Accessed: 2020-05-27.

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

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.

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

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

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

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

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.

David Chase and Yossi Lev. 2005. Dynamic circular work-stealing deque. In *SPAA*. ACM, New York, NY, USA, 21–28. https://doi.org/10.1145/1073970.1073974

Sadegh Dalvandi, Simon Doherty, Brijesh Dongol, and Heike Wehrheim. 2020. Owicki-Gries Reasoning for C11 RAR. In *ECOOP*, Vol. 166. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 11:1–11:26. https://doi.org/10.4230/LIPIcs.ECOOP.2020.11

Hoang-Hai Dang, Jacques-Henri Jourdan, Jan-Oliver Kaiser, and Derek Dreyer. 2019. RustBelt meets relaxed memory. *Proc. ACM Program. Lang.* 4, POPL, Article 34 (Dec. 2019), 29 pages. https://doi.org/10.1145/3371102

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

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

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

Marko Doko and Viktor Vafeiadis. 2017. Tackling real-life relaxed concurrency with FSL++. In *ESOP*. Springer Berlin Heidelberg, Berlin, Heidelberg, 448–475.

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

Shaked Flur, Kathryn E. Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell. 2016. Modelling the ARMv8 architecture, operationally: concurrency and ISA. In *POPL*. ACM, New York, 608–621. https://doi.org/10.1145/2837614.2837615

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

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

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

Jeehoon Kang. 2018. Deque proof. Retrieved June 11, 2020 from https://github.com/jeehoonkang/crossbeam-rfcs/blob/deque-proof/text/2018-01-07-deque-proof.md

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

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

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

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

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

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

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

Roy Margalit and Ori Lahav. 2020. Verifying observational robustness against a C11-style memory model. https://www.cs.tau.ac.il/~orilahav/papers/popl21_robustness_full.pdf (full version of this paper).

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

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

Anton Podkopaev, Ori Lahav, and Viktor Vafeiadis. 2019. Bridging the Gap Between Programming Languages and Hardware Weak Memory Models. *PACMPL* 3, POPL, Article 69 (Jan. 2019), 31 pages. https://doi.org/10.1145/3290382

Rust. 2020. std::sync::Arc. Retrieved June 11, 2020 from https://doc.rust-lang.org/src/alloc/sync.rs.html#213-216

Matthew D. Sinclair, Johnathan Alsop, and Sarita V. Adve. 2017. Chasing away RAts: semantics and evaluation for relaxed atomics on heterogeneous systems. In *ISCA*. ACM, New York, NY, USA, 161–174. https://doi.org/10.1145/3079856.3080206

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

Dmitriy V'jukov. 2008. C++ atomics and memory ordering. Retrieved June 23, 2020 from https://bartoszmilewski.com/2008/12/01/c-atomics-and-memory-ordering/#comment-111

Dmitriy V'jukov. 2013. Relacy race detector. Retrieved July 06, 2020 from http://www.1024cores.net/home/relacy-race-detector

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

Anthony Williams. 2010. Implementing Dekker's algorithm with fences. Retrieved June 08, 2020 from https://www.justsoftwaresolutions.co.uk/threading/implementing_dekkers_algorithm_with_fences.html