Making Weak Memory Models Fair

ORI LAHAV, Tel Aviv University, Israel
EGOR NAMAKONOV, St. Petersburg University, Russia and JetBrains Research, Russia
JONAS OBERHAUSER, Huawei Dresden Research Center, Germany and Huawei OS Kernel Lab, Germany
ANTON PODKOPAEV, HSE University, Russia and JetBrains Research, Russia
VIKTOR VAFEIADIS, MPI-SWS, Germany

Liveness properties, such as termination, of even the simplest shared-memory concurrent programs under sequential consistency typically require some fairness assumptions about the scheduler. Under weak memory models, we observe that the standard notions of *thread fairness* are insufficient, and an additional fairness property, which we call *memory fairness*, is needed.

In this paper, we propose a uniform definition for memory fairness that can be integrated into any declarative memory model enforcing acyclicity of the union of the program order and the reads-from relation. For the well-known models, SC, x86-TSO, RA, and StrongCOH, that have equivalent operational and declarative presentations, we show that our declarative memory fairness condition is equivalent to an intuitive model-specific operational notion of memory fairness, which requires the memory system to fairly execute its internal propagation steps. Our fairness condition preserves the correctness of local transformations and the compilation scheme from RC11 to x86-TSO, and also enables the first formal proofs of termination of mutual exclusion lock implementations under declarative weak memory models.

CCS Concepts: • **Theory of computation** → **Parallel computing models**; **Program semantics**; *Logic and verification.*

Additional Key Words and Phrases: Formal semantics, weak memory models, concurrency, verification

ACM Reference Format:
Ori Lahav, Egor Namakonov, Jonas Oberhauser, Anton Podkopaev, and Viktor Vafeiadis. 2021. Making Weak Memory Models Fair. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 98 (October 2021), 27 pages. https://doi.org/10.1145/3485475

1 INTRODUCTION

Suppose we want to prove termination of a concurrent program under a full-featured weak memory model, such as RC11 [Lahav et al. 2017]. Sadly, this is not currently possible because RC11 does not support reasoning about liveness. Extending its formal definition to enable reasoning about liveness properties is very important because, as shown by Oberhauser et al. [2021a, Table 2], multiple existing mutual exclusion lock implementations hang if too few fences are used. This is also the case for the published version of the HMCS algorithm [Chabbi et al. 2015]: it contains such a termination bug, a simplified version of which we describe in §5.3.

Authors' addresses: Ori Lahav, Tel Aviv University, Israel, orilahav@tau.ac.il; Egor Namakonov, St. Petersburg University, Russia and JetBrains Research, Russia, egor.namakonov@jetbrains.com; Jonas Oberhauser, Huawei Dresden Research Center, Germany and Huawei OS Kernel Lab, Germany, jonas.oberhauser@huawei.com; Anton Podkopaev, HSE University, Russia and JetBrains Research, Russia, apodkopaev@hse.ru; Viktor Vafeiadis, MPI-SWS, Germany, viktor@mpi-sws.org.

2475-1421/2021/10-ART98
https://doi.org/10.1145/3485475

Termination of concurrent programs typically relies on some fairness assumptions about concurrency as illustrated by the following program, whose variables are initialized with 0.

$$x := 1 \,||\, \textbf{repeat}\ \{\ a := x\ \}\ \textbf{until}\ (a \neq 0) \qquad \text{(SpinLoop)}$$

Under *sequential consistency* (SC), the program can diverge if, *e.g.,* thread 2 is always scheduled and thread 1 never gets a chance to run. This run is considered unfair because although thread 1 is always available to be scheduled, it is never selected. A standard assumption is *thread fairness* (which is typically simply called fairness in the literature [Francez 1986; Lamport 1977; Lehmann et al. 1981; Park 1979]), namely that every (unblocked) non-terminated thread is eventually scheduled. With a fair scheduler, SpinLoop is guaranteed to terminate.

Under weak memory consistency, thread fairness alone does not suffice to ensure termination of SpinLoop because merely executing the $x := 1$ write does not mean that its effect is propagated to the other threads. Take, for example, the operational TSO model [Owens et al. 2009], where writes are appended to a thread-local buffer and are later asynchronously applied to the shared memory. With such a model, it is possible that the $x := 1$ write is forever stuck in the first thread's buffer and so thread 2 never gets a chance to read $x = 1$. To rule out such behaviors, we introduce another property, *memory fairness* (MF), that ensures that threads do not indefinitely observe the same stale memory state.

Operational models can easily be extended to support MF by requiring fairness of the internal transitions of the model, which correspond to the propagation of writes to the different threads. For the standard interleaving semantics of SC [Lamport 1979], MF holds vacuously (because the model does not have any internal transitions). For the usual TSO operational model [Owens et al. 2009], MF requires that every buffered write eventually propagates to the main memory. For the operational characterization of *release-acquire* (RA) following Kang et al. [2017], more adaptations are necessary: (1) we constrain the timestamp ordering so that no write can overtake infinitely many other writes; and (2) add a transition that forcefully updates the views of threads so that all executed writes eventually become globally visible. The same criteria are required for MF in the model of *strong coherence* (StrongCOH), which is essentially a restriction of the promise-free fragment of Kang et al. [2017]'s model (as well as of RC11) to relaxed accesses.

In contrast, it is quite challenging to support MF in declarative (a.k.a. axiomatic) models, which have become the norm for hardware architectures (x86-TSO [Owens et al. 2009], Power [Alglave et al. 2014], Arm [Pulte et al. 2017]) and programming languages (e.g., RC11 [Lahav et al. 2017], OCaml [Dolan et al. 2018], JavaAtomics [Bender and Palsberg 2019], Javascript [Watt et al. 2020], WebAssembly [Watt et al. 2019]) alike. In these models, there are no explicit write propagation transitions so that MF could require them to eventually take place. Further, the memory accesses of different threads are not even totally ordered, so even the concept of an event eventually happening is not immediate. We observe, however, neither internal transitions nor a total order are necessary for defining fairness; what is important is that every event is preceded by only a finite number of other events, and this can be defined on the execution graphs used by declarative models.

Specifically, for declarative models satisfying $(\mathsf{po} \cup \mathtt{rf})$-acyclicity (*i.e.,* acyclicity of the union of the program order and the reads-from relation), such as RC11, SC, TSO, RA, and StrongCOH, we show that MF can be defined in a uniform fashion as prefix-finiteness of the extended coherence order. The latter is a relation used in declarative models to order accesses to the same location for guaranteeing SC-per-location [Alglave et al. 2014]. Requiring this relation to be prefix-finite means that in a fair execution no write can be preceded by an infinite number of other events in this order (*e.g.,* reads that have not yet observed the write).

We justify the uniform declarative definition of memory fairness in three ways. First, we show that our declarative MF condition is equivalent to operational MF for models that have equivalent

declarative and operational presentations (*i.e.,* SC, TSO, RA, and StrongCOH). This requires extending the existing equivalence results between operational and declarative models to *infinite* executions, and involves more advanced constructions that make use of memory fairness. Second, we show that including our MF condition in the RC11 declarative language model, which currently lacks any fairness guarantees, incurs no performance overhead: the correctness of local program transformations and the compilation scheme to TSO are unaffected. Third, we show that memory fairness allows lifting robustness theorems about finite executions to infinite ones.

We finally demonstrate that our declarative MF condition enables verification of liveness properties of concurrent programs under RC11 by verifying termination and/or fairness of multiple lock implementations (see §5), including the MCS lock once the fence missing in the presentation of Chabbi et al. [2015] is added. Key to those proofs is a reduction theorem we show for the termination of spinloops. Under certain conditions about the program, which hold for multiple standard implementations, a spinloop terminates under a fair model if and only if it exits whenever an iteration reads only the latest writes in the coherence order. For example, the loop in SpinLoop terminates because reading the latest write ($x := 1$) exits the loop.

Outline. In §2 we define fairness operationally and incorporate it in the operational definitions of SC, x86-TSO, RA, and StrongCOH. In §3 we recap the declarative framework for defining memory models. In §4 we present our declarative MF condition; we establish its equivalence to the operational MF notions and show that it preserves the existing compilation and optimization results for RC11 and that it allows lifting of robustness theorems to infinite executions. In §5 we show that the declarative fairness characterization yields an effective method for proving (non-)termination of spinloops and illustrate it to prove deadlock-freedom and/or fairness of three lock implementations. We conclude with a discussion of fairness in other models in §6.

Supplementary Material. Our technical appendix [Lahav et al. 2021a] contains typeset proofs for the lemmas and propositions of the article. We also provide a Coq development [Lahav et al. 2021b] containing:

- a formalization of operational and declarative fairness for SC, TSO, RA, and StrongCOH;
- proofs of the aforementioned definitions' equivalence (Theorem 4.5);
- a proof of Theorem 5.3 stating a sufficient loop termination condition;
- proofs of termination of the spinlock client and of progress of the ticket lock client for all models satisfying "SC per location" property (which generalizes Theorems 5.4 and 5.5) and of termination of the MCS lock client for SC, TSO and RA (Theorem 5.6 without the RC11 part); and
- a proof of infinite robustness property (Corollary 4.16, excluding the RC11 case).

2 WHAT IS A FAIR OPERATIONAL SEMANTICS?

In this section, we define our operational framework and its fairness constraints. We initially demonstrate our terminology for sequential consistency (SC). In Sections 2.1 to 2.3, we instantiate our framework to the total store order (TSO), release/acquire (RA), and strong coherence (StrongCOH) models, and discuss memory fairness in each of these models.

Labeled Transition Systems. Our formal development is based on *labeled transition systems* (LTSs), which we use to represent both programs and operational memory models. We assume that the transition labels of these systems are split between *(externally) observable transition labels* and *silent transition labels*. Using transition labels we define a *trace* to be a (finite or infinite) sequence of transition labels (of any kind); whereas an *observable trace* is a (finite or infinite) sequence of

observable transition labels. Then, LTSs capture sets of traces and observable traces in the standard way, which is formulated below.

Formally, we define an LTS A to be a tuple $\langle Q, \Sigma, \Theta, \mathit{init}, \rightarrow \rangle$, where Q is a set of *states*, Σ is a set of *observable transition labels*, Θ is a set of *silent transition labels*, $\mathit{init} \in Q$ is the *initial state*, and $\rightarrow \subseteq Q \times (\Sigma \uplus \Theta) \times Q$ is a set of *transitions*. We denote by $A.\mathtt{Q}$, $A.\Sigma$, $A.\Theta$, $A.\mathtt{init}$, and $\rightarrow_A$ the components of an LTS A.

We denote by $\mathtt{src}(t)$, $\mathtt{tlab}(t)$, and $\mathtt{tgt}(t)$ the three components of a transition $t \in \rightarrow$. For $\sigma \in \Sigma \uplus \Theta$, we write $\xrightarrow{\sigma}$ for the relation $\{\langle \mathtt{src}(t), \mathtt{tgt}(t) \rangle \mid t \in \rightarrow, \mathtt{tlab}(t) = \sigma\}$. We use $\rightarrow$ for the relation $\bigcup_{\sigma \in \Sigma \uplus \Theta} \xrightarrow{\sigma}$. We say that a transition label $\sigma \in \Sigma \uplus \Theta$ is *enabled* in some state $q \in Q$ if $q \xrightarrow{\sigma} q'$ for some $q' \in Q$.

A *run* of A is a (finite or infinite) sequence μ of transitions in $\rightarrow_A$ such that $\mathtt{src}(\mu(0)) = A.\mathtt{init}$ and $\mathtt{tgt}(\mu(k-1)) = \mathtt{src}(\mu(k))$ for every $k \geq 1$ in $\mathit{dom}(\mu)$. A run μ of A *induces* the trace ρ if $\rho(k) = \mathtt{tlab}(\mu(k))$ for every $k \in \mathit{dom}(\mu)$. Also, μ induces the *observable* trace ρ' if ρ' is the restriction to Σ of some trace ρ that is induced by μ.

An (observable) trace ρ is called an (observable) trace of A if it is induced by some run of A. We write $\mathsf{OTr}(A)$ for the set of all observable traces of A and $\mathsf{OTr}^{\mathsf{fin}}(A)$ for the set of all finite observable traces of A.

Domains and Event Labels. To define programs and their semantics, we fix sets Loc, Tid, and Val of *(shared) locations*, *thread identifiers*, and *values* (respectively). We assume that Val contains a distinguished value 0, which serves as the initial value for all locations. In addition, we assume that Tid is finite, given by $\mathsf{Tid} = \{1, 2, \ldots, N\}$ for some $N \geq 1$. (Our main result below requires Tid to be finite, see Remark 3.) We use x, y to range over Loc; τ, π to range over Tid; and v to range over Val. Programs interact with the memory using *event labels*, defined as follows.

Definition 2.1. An *event label* l is one of the following:

- Read event label: $\mathtt{R}(x, v_{\mathtt{R}})$ where $x \in \mathsf{Loc}$ and $v_{\mathtt{R}} \in \mathsf{Val}$.
- Write event label: $\mathtt{W}(x, v_{\mathtt{W}})$ where $x \in \mathsf{Loc}$ and $v_{\mathtt{W}} \in \mathsf{Val}$.
- Read-modify-write label: $\mathtt{RMW}(x, v_{\mathtt{R}}, v_{\mathtt{W}})$ where $x \in \mathsf{Loc}$ and $v_{\mathtt{R}}, v_{\mathtt{W}} \in \mathsf{Val}$.

The functions $\mathtt{typ}$, $\mathtt{loc}$, $\mathtt{val}_{\mathtt{r}}$, and $\mathtt{val}_{\mathtt{w}}$ return (when applicable) the type (R/W/RMW), location (x), read value ($v_{\mathtt{R}}$), and written value ($v_{\mathtt{W}}$) of a given event label l. We denote by ELab the set of all event labels.

Remark 1. For conciseness, we have not included *fences* in the set of event labels. In TSO [Owens et al. 2009] and RA [Lahav et al. 2016], fences can be modeled as read-modify-writes to an otherwise-unused distinguished location f.

Remark 2. Rich programming languages like C/C++ [Batty et al. 2011] and Java [Bender and Palsberg 2019] as well as the Armv8 multiprocessor [Pulte et al. 2017] have multiple kinds of accesses. This requires us to extend our event labels with additional modifiers. However, simple event labels as defined above suffice for the purpose of this paper.

Sequential Programs. To keep the presentation abstract, we do not fix a particular programming language, but rather represent sequential (thread-local) programs as LTSs with ELab, the set of all event labels, serving as the set of observable transition labels. For simplicity, we assume that sequential programs do not have silent transitions.[1] For an example of a toy programming language syntax and its reading as an LTS, see [Podkopaev et al. 2019]. In our code snippets throughout the paper, we implicitly assume such a standard interpretation.

[1]This assumption serves us merely to simplify the presentation, since silent program transitions can be always attached to the next memory access.

We refer to observable traces of sequential programs (*i.e.,* sequences over ELab) as *sequential traces*.

Example 2.2. The simple sequential program **repeat** { $a := x$ } **until** $(a \neq 0)$ is formally captured as an LTS with an initial state *init* and a state *final*, and transitions $\langle \mathit{init}, \mathtt{R}(x, v), \mathit{init} \rangle$ for every $v \in \mathsf{Val} \setminus \{0\}$ and $\langle \mathit{init}, \mathtt{R}(x, 0), \mathit{final} \rangle$. The sequential traces $\mathtt{R}(x,0), \mathtt{R}(x,0), \mathtt{R}(x,0), \mathtt{R}(x,42)$ is an (observable) trace of this program. The infinite sequential trace $\mathtt{R}(x,0), \mathtt{R}(x,0), \ldots$ is another (observable) trace of this program.

Concurrent Programs. A *concurrent program*, which we also simply call a *program*, is a top-level parallel composition of sequential programs, defined as a *finite* mapping assigning a sequential program to each thread $\tau \in \mathsf{Tid}$. A concurrent program P induces an LTS with $\mathsf{Tid} \times \mathsf{ELab}$ serving as the set of observable transition labels (and no silent transition labels). This LTS follows the interleaving semantics of P: its states are tuples in $\prod_{\tau \in \mathsf{Tid}} P(\tau).\mathtt{Q}$; the initial state is $\lambda\tau.\ P(\tau).\mathtt{init}$; and the transitions are given by:

$$\frac{\overline{p}(\tau) \xrightarrow{l}_{P(\tau)} p}{\overline{p} \xrightarrow{\tau:l}_{P} \overline{p}[\tau \mapsto p]}$$

In the sequel, we identify concurrent programs with their induced LTSs.

We refer to observable traces of concurrent programs (*i.e.,* sequences over $\mathsf{Tid} \times \mathsf{ELab}$) as *concurrent traces*. We denote the two components of a pair $\sigma \in \mathsf{Tid} \times \mathsf{ELab}$ by $\mathtt{tid}(\sigma)$ and $\mathtt{elab}(\sigma)$ respectively.

Behaviors. We define a *behavior* to be a function β assigning a sequential trace to every thread, since the events executed by each thread capture precisely what it has observed about the memory system.

Notation 2.3. *The restriction of a concurrent trace ρ to thread $\tau \in \mathsf{Tid}$, denoted by $\rho|_\tau$, is the sequence obtained from ρ by keeping only the transition labels of the form $\tau : \_$.*

Definition 2.4. The *behavior induced by a concurrent trace* ρ, denoted by $\beta(\rho)$, is given by

$$\beta(\rho) \triangleq \lambda\tau \in \mathsf{Tid}.\ \lambda k \in \mathit{dom}(\rho|_\tau).\ \mathtt{elab}(\rho|_\tau(k)).$$

This notation is extended to sets of concurrent traces in the obvious way ($\beta(S) \triangleq \{\beta(\rho) \mid \rho \in S\}$).

Notation 2.5. *For an LTS A with $A.\Sigma = \mathsf{Tid} \times \mathsf{ELab}$, we denote by $\mathsf{B}(A)$ the set of behaviors induced by observable traces of A (*i.e., $\mathsf{B}(A) \triangleq \beta(\mathsf{OTr}(A))$*) and by $\mathsf{B}^{\mathsf{fin}}(A)$ the set of behaviors induced by finite observable traces of A (*i.e., $\mathsf{B}^{\mathsf{fin}}(A) \triangleq \beta(\mathsf{OTr}^{\mathsf{fin}}(A))$*).*

Since operations of different threads commute in the program semantics, the following property easily follows from our definitions.

Proposition 2.6. *For every program P, if $\beta(\rho_1) = \beta(\rho_2)$, then $\rho_1 \in \mathsf{OTr}(P)$ iff $\rho_2 \in \mathsf{OTr}(P)$.*

Thread Fairness. Not all program behaviors are fair.

Example 2.7. Consider the following program:

$$x := 1 \;\Big\|\; \begin{array}{l} L\colon\ a := x \\ \quad\ \textbf{if}\ a = 0\ \textbf{goto}\ L \end{array} \qquad \text{(Rloop)}$$

The behaviors of this program include the behavior assigning $\mathtt{W}(x,1)$ to the first thread and $\mathtt{R}(x,1)$ to the second, but also the (infinite) behavior assigning the empty sequence to the first thread and

the infinite sequence $\mathtt{R}(x,0), \mathtt{R}(x,0), \ldots$ to the second. This behavior occurs if an unfair scheduler only schedules the second thread to run even though the first thread is always available to execute.[2]

A natural constraint, which in particular excludes the infinite behavior in the example above, requires a fair scheduler. Since our formalism assumes no blocking operations (in particular, locks are implemented using spinloops), such a scheduler has to ensure that every non-terminated thread is eventually scheduled, which we formally define as follows.

Definition 2.8. Let P be a program.

- A thread $\tau \in \mathsf{Tid}$ is *enabled* in $\overline{p} \in P.\mathtt{Q}$ if $\langle \tau, l \rangle$ is enabled in $\overline{p}$ for some $l \in \mathsf{ELab}$.
- A thread $\tau \in \mathsf{Tid}$ is *continuously enabled at index k* in an infinite run μ of P if it is enabled in $\mathtt{src}(\mu(j))$ for every index $j \geq k$. Thread τ is *continuously enabled* in μ if it is continuously enabled in μ at some index k.
- A run μ of P is *thread-fair* if μ is finite or for every thread $\tau \in \mathsf{Tid}$ and index k such that τ is continuously enabled in μ at k, there exists $j \geq k$ such that $\mathtt{tid}(\mathtt{tlab}(\mu(j))) = \tau$.
- A *thread-fair observable trace of P* is any concurrent trace induced by a thread-fair run of P.
- A *thread-fair behavior of P* is any behavior induced by a thread-fair observable trace of P. We denote by $\mathsf{B}^{\mathsf{tf}}(P)$ the set of all thread-fair behaviors of P.

Returning to Example 2.7, thread-fair behaviors of Rloop are either finite or must assign $\mathtt{W}(x,1)$ to the first thread.

Again, since operations of different threads commute in the program semantics, the following property easily follows from our definitions.

Proposition 2.9. *For every program P, if $\beta(\rho_1) = \beta(\rho_2)$, then ρ_1 is a thread-fair observable trace of P iff ρ_2 is a thread-fair observable trace of P.*

Memory Systems. To give operational semantics to programs, we synchronize them with *memory systems*, which, like programs, are LTSs with $\mathsf{Tid} \times \mathsf{ELab}$ serving as the set of observable transition labels. In addition, memory systems have silent transition labels, which vary from one system to another. Intuitively, the set of silent transition labels $\mathcal{M}.\Theta$ of a memory system $\mathcal{M}$ consists of internal actions that the program cannot observe (*e.g.,* cache-related operations).

The most well-known memory system is that of *sequential consistency* [Lamport 1979], denoted here by $\mathcal{M}_{\mathsf{SC}}$, in which writes by each thread are made immediately visible to all other threads. $\mathcal{M}_{\mathsf{SC}}$ tracks the most recent value written to each location. Its initial state maps each location to zero. That is, $\mathcal{M}_{\mathsf{SC}}.\mathtt{Q} \triangleq \mathsf{Loc} \to \mathsf{Val}$ and $\mathcal{M}_{\mathsf{SC}}.\mathtt{init} \triangleq \lambda x.\ 0$. The system $\mathcal{M}_{\mathsf{SC}}$ has no silent transitions ($\mathcal{M}_{\mathsf{SC}}.\Theta = \emptyset$) and its transition relation $\rightarrow_{\mathcal{M}_{\mathsf{SC}}}$ is defined as follows:

$$\frac{M' = M[x \mapsto v]}{M \xrightarrow{\tau:\mathtt{W}(x,v)}_{\mathcal{M}_{\mathsf{SC}}} M'} \qquad \frac{M(x) = v}{M \xrightarrow{\tau:\mathtt{R}(x,v)}_{\mathcal{M}_{\mathsf{SC}}} M} \qquad \frac{M \xrightarrow{\tau:\mathtt{R}(x,v_{\mathtt{R}})}_{\mathcal{M}_{\mathsf{SC}}} \xrightarrow{\tau:\mathtt{W}(x,v_{\mathtt{W}})}_{\mathcal{M}_{\mathsf{SC}}} M'}{M \xrightarrow{\tau:\mathtt{RMW}(x,v_{\mathtt{R}},v_{\mathtt{W}})}_{\mathcal{M}_{\mathsf{SC}}} M'}$$

Writing v to x simply updates the value of x stored in M. ($M[x \mapsto v]$ is the function that maps x to v and all other locations y to $M(y)$.) Reading v from x succeeds iff the value stored for x in memory is v. The atomic read-modify-write $\mathtt{RMW}(x, v_{\mathtt{R}}, v_{\mathtt{W}})$ reads location x yielding value $v_{\mathtt{R}}$ and immediately writes $v_{\mathtt{W}}$ to it. Note that $\mathcal{M}_{\mathsf{SC}}$ is oblivious to the thread that takes the action ($\xrightarrow{\tau:l}_{\mathcal{M}_{\mathsf{SC}}} = \xrightarrow{\pi:l}_{\mathcal{M}_{\mathsf{SC}}}$). The other memory systems below do not have this property.

[2] On this level, without considering a particular memory system (as defined below), the read values are not restricted whatsoever. Thus, the behaviors of this program include also any behavior assigning $\mathtt{W}(x,1)$ to the first thread and either $\mathtt{R}(x,v)$ for some $v \in \mathsf{Val} \setminus \{0\}$ or the infinite sequence $\mathtt{R}(x,0), \mathtt{R}(x,0), \ldots$ to the second thread. Nonsensical behaviors (with $v \notin \{0,1\}$) are overruled when the program is linked with any of the memory systems defined below, with or without "memory fairness".

Linking Programs and Memory Systems. By linking programs and memory systems, we can talk about the behavior of a program P under a memory system $\mathcal{M}$. We say that a certain behavior β is a *behavior of a program P under a memory system $\mathcal{M}$* if β is both a behavior of P and a behavior of $\mathcal{M}$ (*i.e.,* $\beta \in \mathsf{B}(P) \cap \mathsf{B}(\mathcal{M})$). Similarly, β is called a *thread-fair behavior of P under $\mathcal{M}$* if $\beta \in \mathsf{B}^{\mathsf{tf}}(P) \cap \mathsf{B}(\mathcal{M})$.

Proposition 2.10. *Let P be a program, $\mathcal{M}$ be a memory system, and β be a behavior.*

- *β is a behavior of P under $\mathcal{M}$ iff $\beta = \beta(\rho)$ for some $\rho \in \mathsf{OTr}(P) \cap \mathsf{OTr}(\mathcal{M})$.*
- *β is a thread-fair behavior of P under $\mathcal{M}$ iff $\beta = \beta(\rho)$ for some $\rho \in \mathsf{OTr}(\mathcal{M})$ that is also a thread-fair observable trace of P.*

Example 2.11. Thread-fair behaviors of the program Rloop under $\mathcal{M}_{\mathsf{SC}}$ must be finite. Indeed, in observable traces of $\mathcal{M}_{\mathsf{SC}}$, after the first thread performs $\mathtt{W}(x, 1)$, the second thread will perform $\mathtt{R}(x, 1)$ and terminate its execution. The behavior β_{inf} that assigns the empty sequence to the first thread and the infinite sequence consisting of $\mathtt{R}(x, 0)$ event labels to the second thread cannot be obtained from a thread-fair run of Rloop.

Memory Fairness. As we have already discussed, thread-fairness alone is often insufficient to reason about termination under weak memory models. For this reason, we introduce *memory fairness* (MF), which ensures that a thread cannot be lagging behind indefinitely because the memory system did not propagate certain updates to it. We formalize this intuition by having MF require that the memory silent transitions (responsible for such propagation steps) are scheduled infinitely often.

Definition 2.12. Let $\mathcal{M}$ be a memory system.

- A silent transition label $\theta \in \mathcal{M}.\Theta$ is *continuously enabled at index k* in an infinite run μ of $\mathcal{M}$ if it is enabled in $\mathtt{src}(\mu(j))$ for every index $j \geq k$. The label θ is *continuously enabled* in μ if it is continuously enabled in μ at some index k.
- A run μ of $\mathcal{M}$ is *memory-fair* if μ is finite or for every silent memory transition label $\theta \in \mathcal{M}.\Theta$ and index k such that θ is continuously enabled in μ at k, there exists $j \geq k$ such that $\mathtt{tlab}(\mu(j)) = \theta$.
- A *memory-fair observable trace of $\mathcal{M}$* is any concurrent trace induced by a memory-fair run of $\mathcal{M}$.
- A *memory-fair behavior of $\mathcal{M}$* is any behavior induced by a memory-fair observable trace of $\mathcal{M}$. We denote by $\mathsf{B}^{\mathsf{mf}}(\mathcal{M})$ the set of all memory-fair behaviors of $\mathcal{M}$.

Linking this definition with programs, we say that a certain behavior β is a *memory-fair behavior of a program P under a memory system $\mathcal{M}$* if $\beta \in \mathsf{B}(P) \cap \mathsf{B}^{\mathsf{mf}}(\mathcal{M})$. Similarly, β is called a *thread&memory-fair behavior of P under $\mathcal{M}$* if $\beta \in \mathsf{B}^{\mathsf{tf}}(P) \cap \mathsf{B}^{\mathsf{mf}}(\mathcal{M})$.

Proposition 2.13. *Let P be a program, $\mathcal{M}$ be a memory system, and β be a behavior.*

- *β is a memory-fair behavior of P under $\mathcal{M}$ iff $\beta = \beta(\rho)$ for some observable trace ρ of P that is also a memory-fair observable trace of $\mathcal{M}$.*
- *β is a thread&memory-fair behavior of P under $\mathcal{M}$ iff $\beta = \beta(\rho)$ for some thread-fair observable trace ρ of P that is also a memory-fair observable trace of $\mathcal{M}$.*

Since $\mathcal{M}_{\mathsf{SC}}.\Theta = \emptyset$, every behavior of a program P under $\mathcal{M}_{\mathsf{SC}}$ is (vacuously) memory-fair.

Example 2.14. Consider the following program (assuming that x is initialized to 0):

$$\begin{array}{l} L_1\colon x := 1 \\ \quad\ x := 0 \\ \quad\ \textbf{goto}\ L_1 \end{array} \Bigg\| \begin{array}{l} L_2\colon a := x \\ \quad\ \textbf{if}\ a = 0\ \textbf{goto}\ L_2 \end{array} \qquad \text{(WWRloop)}$$

$$\frac{B' = B[\tau \mapsto \langle x, v\rangle \cdot B(\tau)]}{M, B \xrightarrow{\tau:\mathtt{W}(x,v)}_{\mathcal{M}_{\mathrm{TSO}}} M, B'} \qquad \frac{B(\tau) = \langle x_n, v_n\rangle \cdot \ldots \cdot \langle x_1, v_1\rangle \quad M[x_1 \mapsto v_1] \cdots [x_n \mapsto v_n](x) = v}{M, B \xrightarrow{\tau:\mathtt{R}(x,v)}_{\mathcal{M}_{\mathrm{TSO}}} M, B}$$

$$\frac{B(\tau) = \epsilon \qquad M(x) = v_{\mathsf{R}}}{M, B \xrightarrow{\tau:\mathtt{RMW}(x,v_{\mathsf{R}},v_{\mathsf{W}})}_{\mathcal{M}_{\mathrm{TSO}}} M[x \mapsto v_{\mathsf{W}}], B} \qquad \frac{B(\tau) = b \cdot \langle x, v\rangle}{M, B \xrightarrow{\mathtt{prop}(\tau)}_{\mathcal{M}_{\mathrm{TSO}}} M[x \mapsto v], B[\tau \mapsto b]}$$

Fig. 1. Transitions of $\mathcal{M}_{\mathrm{TSO}}$

The infinite behavior that assigns the infinite sequences $\mathtt{W}(x,1), \mathtt{W}(x,0), \mathtt{W}(x,1), \mathtt{W}(x,0), \ldots$, and $\mathtt{R}(x,0), \mathtt{R}(x,0), \ldots$ to the first and second threads (respectively) is a thread&memory-fair behavior of this program under $\mathcal{M}_{\mathrm{SC}}$: in a corresponding run both threads are executed infinitely often. In particular, note that our definitions require that transitions that are *continuously* enabled are eventually taken, and while the transition $\mathtt{R}(x,1)$ is infinitely often enabled for the second thread, it is not continuously enabled.

Next, we demonstrate three weaker memory systems with non-empty sets of silent transitions that have non-memory-fair traces. In these systems, whether a program terminates or deadlocks may crucially depend on memory fairness.

2.1 The Total Store Order Memory System

We instantiate memory fairness to the "Total Store Order" (TSO) model [Owens et al. 2009; Sewell et al. 2010] of the x86 architecture. This memory system, denoted by $\mathcal{M}_{\mathrm{TSO}}$, is defined by:

(1) $\mathcal{M}_{\mathrm{TSO}}.\mathtt{Q} \triangleq (\mathsf{Loc} \to \mathsf{Val}) \times (\mathsf{Tid} \to (\mathsf{Loc} \times \mathsf{Val})^*)$
(Each state consists of a memory and a per-thread store buffer.)
(2) $\mathcal{M}_{\mathrm{TSO}}.\Theta \triangleq \{\mathtt{prop}(\tau) \mid \tau \in \mathsf{Tid}\}$
(Silent transitions consist of a propagation label for every thread.)
(3) $\mathcal{M}_{\mathrm{TSO}}.\mathtt{init} \triangleq \langle M_0, B_0\rangle$, where $M_0 \triangleq \lambda x.\ 0$ and $B_0 \triangleq \lambda \tau.\ \epsilon$ (Initially, all buffers are empty.)
(4) $\to_{\mathcal{M}_{\mathrm{TSO}}}$ is given in Fig. 1.

In addition to the global memory M, states of $\mathcal{M}_{\mathrm{TSO}}$ include a mapping B assigning a FIFO *store buffer* to every thread. Writes are first written to the local buffer and later non-deterministically propagate to memory (in the order in which they were issued). Reads read the most recent value of the relevant location in the thread's buffer and refer to the memory if such value does not exist. RMWs can only execute when the thread's buffer is empty and write their result in the memory directly.

Example 2.15 (Store Buffering). The following annotated behavior is *allowed* under $\mathcal{M}_{\mathrm{TSO}}$ (but not under $\mathcal{M}_{\mathrm{SC}}$):

$$\begin{array}{l|\!|l} x := 1 & y := 1 \\ a := y \ \textit{// reads 0} & a := x \ \textit{// reads 0} \end{array} \tag{SB}$$

Indeed, the first thread may run first, but the write of 1 to x may remain in its store buffer. Then, when the second thread runs, it reads the initial value (0) of x from the memory.

Example 2.16. Revisiting the Rloop program from §2, unlike under $\mathcal{M}_{\mathrm{SC}}$, thread-fair behaviors of Rloop under $\mathcal{M}_{\mathrm{TSO}}$ include the (infinite) behavior assigning the $\mathtt{W}(x,1)$ to the first thread and the infinite sequence $\mathtt{R}(x,0), \mathtt{R}(x,0), \ldots$ to the second. Indeed, the entry $\langle x, 1\rangle$ may indefinitely remain in the first thread's buffer, so that $\mathtt{W}(x,1)$ is never executed from the point of view of the second

thread. To disqualify this behavior, we need to further require *memory* fairness. Indeed, in runs inducing this infinite behavior, the silent memory transition $\texttt{prop}(1)$ is necessarily continuously enabled. Memory fairness requires that $\texttt{prop}(1)$ will be eventually executed, and from that point on $\mathcal{M}_{\mathsf{TSO}}$ prohibits the second thread from executing $\texttt{R}(x, 0)$.

We note that the notion of memory fairness is sensitive to the choice of silent memory transitions. For example, consider an alternative memory system, denoted by $\mathcal{M}'_{\mathsf{TSO}}$, with less informative silent transition labels that do not record the thread identifier of the propagated write. (Formally $\mathcal{M}'_{\mathsf{TSO}}$ is defined just like $\mathcal{M}_{\mathsf{TSO}}$ except for $\mathcal{M}'_{\mathsf{TSO}}.\Theta \triangleq \{\texttt{prop}\}$, and the label of the propagation step is $\texttt{prop}$ rather than $\texttt{prop}(\tau)$.) Then, $\mathcal{M}'_{\mathsf{TSO}}$ induces the same set of behaviors as $\mathcal{M}_{\mathsf{TSO}}$, but not the same set of *memory fair* behaviors. In particular, we can extend the Rloop program with an additional thread that constantly writes to some unrelated location y, and obtain a memory fair run of $\mathcal{M}'_{\mathsf{TSO}}$ by infinitely often propagating a write to y, but never propagating the $\texttt{W}(x, 1)$ entry.

2.2 The Release/Acquire Memory System

We instantiate our operational framework with a memory system for Release/Acquire (RA), enriched with silent memory transitions for capturing fair behaviors. Here we follow an operational formulation of RA from Kaiser et al. [2017], based on the Promising Semantics of Kang et al. [2017].

The memory of the RA system records a (finite) set of *messages*, each of which corresponds to some write that was previously executed. Messages (of the same location) are ordered using *timestamps*, and carry a *view*—a mapping from locations to timestamps. In turn, the states of this memory system also keep track of the current view of each thread, and use these views to confine the set of messages that threads may read and write. In particular, if a thread has observed (either by reading or by writing itself) a message whose view V has $V(x) = t$, then it can only read messages of x whose timestamp is greater than or equal to t.

To formally define this system, we let $\mathsf{Time} \triangleq \mathbb{N}$ (using natural numbers as timestamps), $\mathsf{View} \triangleq \mathsf{Loc} \to \mathsf{Time}$ (the set of views), and $\mathsf{Msg} \triangleq \mathsf{Loc} \times \mathsf{Val} \times \mathsf{Time} \times \mathsf{View}$ (the set of messages). We denote a message m as a tuple of the form $\langle x : v@t, V\rangle$, where $x \in \mathsf{Loc}$, $v \in \mathsf{Val}$, $t \in \mathsf{Time}$, and $V \in \mathsf{View}$. We write $\texttt{loc}(m)$, $\texttt{val}(m)$, $\texttt{ts}(m)$, and $\texttt{view}(m)$ to refer to the components of a message m. The usual order $<$ on natural numbers is lifted pointwise to a partial order on views; $\sqcup$ denotes the pointwise maximum on views; and V_0 is the minimum view ($V_0 \triangleq \lambda x.\ 0$).

With these definitions and notations, the RA memory system, denoted here by $\mathcal{M}_{\mathsf{RA}}$, is defined as follows (additional silent memory transitions are discussed below):

(1) $\mathcal{M}_{\mathsf{RA}}.\texttt{Q} \triangleq \mathcal{P}(\mathsf{Msg}) \times (\mathsf{Tid} \to \mathsf{View})$.
(2) $\mathcal{M}_{\mathsf{RA}}.\texttt{init} \triangleq \langle M_0, \lambda\tau.\ V_0\rangle$, where the initial memory is $M_0 \triangleq \{\langle x : 0@0, V_0\rangle \mid x \in \mathsf{Loc}\}$.
(3) $\to_{\mathcal{M}_{\mathsf{RA}}}$ is given in Fig. 2.

The states of $\mathcal{M}_{\mathsf{RA}}$ consist of a set M of all messages added to the memory so far and a mapping T assigning a view to each thread. Write steps of thread τ writing to location x pick a timestamp t that is fresh ($\nexists m \in M.\ \texttt{loc}(m) = x \wedge \texttt{ts}(m) = t$) and greater than the latest timestamp that τ has observed for x ($T(\tau)(x) < t$); update the thread's view to include this timestamp ($T' = T[\tau \mapsto T(\tau)[x \mapsto t]]$); and add a corresponding message to the memory carrying the (updated) thread view ($M' = M \cup \{\langle x : v@t, T'(\tau)\rangle\}$). Read steps of thread τ reading from location x pick a message from the current memory ($\langle x : v@t, V\rangle \in M$) whose timestamp is greater than or equal to the latest timestamp that τ has observed for x ($T(\tau)(x) \le t$); and incorporate the message's view in the thread view ($T' = T[\tau \mapsto T(\tau) \sqcup V]$). RMW steps are defined as atomic sequencing of a read step followed by a write step, with the restriction that the new message's (fresh) timestamp is the successor of the timestamp of the read message ($T''(\tau)(x) = T'(\tau)(x) + 1$). The latter condition is needed to ensure the atomicity of RMWs: no other write can intervene between the read part

$$\frac{\nexists m \in M.\ \mathtt{loc}(m) = x \wedge \mathtt{ts}(m) = t \quad T(\tau)(x) < t \quad T' = T[\tau \mapsto T(\tau)[x \mapsto t]] \quad M' = M \cup \{\langle x : v@t, T'(\tau)\rangle\}}{\langle M, T\rangle \xrightarrow{\tau:\mathtt{W}(x,v)}_{\mathcal{M}_{\mathsf{RA}}} \langle M', T'\rangle} \qquad \frac{\langle x : v@t, V\rangle \in M \quad T(\tau)(x) \le t \quad T' = T[\tau \mapsto T(\tau) \sqcup V]}{\langle M, T\rangle \xrightarrow{\tau:\mathtt{R}(x,v)}_{\mathcal{M}_{\mathsf{RA}}} \langle M, T'\rangle}$$

$$\frac{\langle M, T\rangle \xrightarrow{\tau:\mathtt{R}(x,v_{\mathtt{R}})}_{\mathcal{M}_{\mathsf{RA}}} \langle M, T'\rangle \xrightarrow{\tau:\mathtt{W}(x,v_{\mathtt{W}})}_{\mathcal{M}_{\mathsf{RA}}} \langle M', T''\rangle \qquad T''(\tau)(x) = T'(\tau)(x) + 1}{\langle M, T\rangle \xrightarrow{\tau:\mathtt{RMW}(x,v_{\mathtt{R}},v_{\mathtt{W}})}_{\mathcal{M}_{\mathsf{RA}}} \langle M', T''\rangle}$$

Fig. 2. Transitions of $\mathcal{M}_{\mathsf{RA}}$

and the write part of the RMW (*i.e.,* no message can be placed between the read and the written messages in the timestamp order).

Example 2.17 (Message passing). The following annotated behavior is *disallowed* under $\mathcal{M}_{\mathsf{RA}}$:

$$\begin{array}{l|l} x := 1 & a := y \ \textit{// reads 1} \\ y := 1 & b := x \ \textit{// reads 0} \end{array} \tag{MP}$$

Indeed, the second thread can read 1 for y, only after the first thread added two messages $m_x = \langle x : 1@t_x, [x \mapsto t_x]\rangle$ and $m_y = \langle y : 1@t_y, [x \mapsto t_x, y \mapsto t_y]\rangle$ to the memory with $t_x, t_y > 0$. When reading m_y, the second thread increases its view of x to be t_x. Since $t_x > 0$, it is then unable to read the initial message of x, and must read m_x.

Example 2.18. By forcing RMWs to use the successor of the read message as the timestamp of the written message, $\mathcal{M}_{\mathsf{RA}}$ forbids different RMWs to read the same message. To see this, consider the following example (where **FADD** denotes an atomic fetch-and-add instruction):

$$a := \mathbf{FADD}(x, 1)\ \textit{// reads 0} \ \Big\|\ b := \mathbf{FADD}(x, 1)\ \textit{// reads 0} \tag{2RMW}$$

W.l.o.g., if the first runs first, it reads from the initialization message $\langle x : 0@0, V_0\rangle$ (it is the only message of x in M_0), and it is forced to add a message *with timestamp* 1. When the second thread runs, it may *not* read from the initialization message: that would again require adding a message of x with timestamp 1, but that timestamp is no longer available. Thus, it may only read from the message that was added by the first thread.

Example 2.19. Fences (modeled as RMWs to an otherwise unused distinguished location f) can be used to recover sequential consistency when needed. The following outcome is forbidden by RA.

$$\begin{array}{l|l} x := 1 & y := 1 \\ \mathbf{FADD}(f, 0) & \mathbf{FADD}(f, 0) \\ a := y \ \textit{// reads 0} & b := x \ \textit{// reads 0} \end{array} \tag{SB+RMWs}$$

Due to the RMWs in both threads, $\mathcal{M}_{\mathsf{RA}}$ forbids the annotated program behavior. Indeed, suppose, w.l.o.g., that the first thread executes its $\mathbf{FADD}(f, 0)$ first, it will read from the initialization message to f and will add to memory a message of the form $\langle f : 0@1, V\rangle$ with $V(x) > 0$. When the second thread executes its $\mathbf{FADD}(f, 0)$, it will necessarily read that message and incorporate the view V in its thread view, so that its view of x will be increased. Then, when it reads x it may not pick the initial message.

The RA memory system defined so far (with no silent transitions) allows non-fair executions. In particular, it allows messages added by some thread to never propagate to other threads, so that other threads may forever read a message with a lower timestamp, and thus, allows, *e.g.,* a thread-fair *infinite* behavior for the Rloop program from §2.

To address this problem, we include silent memory transitions in $\mathcal{M}_{\mathsf{RA}}$, labeled with tuples of the form $\mathtt{prop}(\tau, m)$, where $\tau \in \mathsf{Tid}$ and $m \in \mathsf{Msg}$ (*i.e.,* $\mathcal{M}_{\mathsf{RA}}.\Theta \triangleq \{\mathtt{prop}(\tau, m) \mid \tau \in \mathsf{Tid}, m \in \mathsf{Msg}\}$). Then, we include in $\mathcal{M}_{\mathsf{RA}}$ the following silent memory step:

$$\textsc{RA-propagate}\quad \frac{m \in M \qquad T(\tau)(\mathtt{loc}(m)) < \mathtt{ts}(m)}{\langle M, T\rangle \xrightarrow{\mathtt{prop}(\tau,m)}_{\mathcal{M}_{\mathsf{RA}}} \langle M, T[\tau \mapsto T(\tau)[\mathtt{loc}(m) \mapsto \mathtt{ts}(m)]]\rangle}$$

For a given thread τ and message m that has not been yet observed by thread τ ($T(\tau)(\mathtt{loc}(m)) < \mathtt{ts}(m)$), this step increases τ's view to include m's timestamp. Intuitively speaking, it ensures that every thread τ eventually advances its view so that it cannot keep reading an old message indefinitely.

Example 2.20. While thread-fair behaviors of Rloop under $\mathcal{M}_{\mathsf{RA}}$ include an infinite behavior (in which the second thread indefinitely read the initialization message), memory fairness forbids this behavior. Indeed, in runs inducing this infinite behavior, a silent label $\mathtt{prop}(2, \langle x : 1@t, [x \mapsto t]\rangle)$ (where t is a timestamp of a message added by instruction $x := 1$ of Rloop) is necessarily continuously enabled. Memory fairness ensures that the corresponding transition is eventually executed, and from that point on, $\mathcal{M}_{\mathsf{RA}}$ prohibits the second thread from executing $\mathtt{R}(x, 0)$.

We emphasize again that memory fairness is sensitive to the choice of silent memory transitions. For instance, the system obtained from $\mathcal{M}_{\mathsf{RA}}$ by discarding the message m from the labels of silent memory steps induces the same set of behaviors as $\mathcal{M}_{\mathsf{RA}}$, but not the same set of *memory fair* behaviors. In the next sections, we present the declarative approach for defining the semantics of memory systems, which uniformly captures memory fairness, and does not require the technical ingenuity needed for ensuring fairness in operational memory systems.

2.3 The Strong-Coherence Memory System

We consider a memory system for Strong-Coherence (StrongCOH), *i.e.,* the relaxed fragment of RC11. Similar to RA, we follow an operational formulation of StrongCOH following the relaxed and promise-free fragment of the Promising Semantics of Kang et al. [2017]. Since this formulation is very close to RA's one discussed above, we describe only the difference between them.

The states of $\mathcal{M}_{\mathsf{StrongCOH}}$ are the same as of $\mathcal{M}_{\mathsf{RA}}$, and transitions are similar, where the only difference is in the read transition (note the crossed out "$\sqcup V$"):

$$\frac{\langle x : v@t, V\rangle \in M \qquad T(\tau)(x) \le t \qquad T' = T[\tau \mapsto T(\tau)[x \mapsto t] \not\sqcup V]}{\langle M, T\rangle \xrightarrow{\tau:\mathtt{R}(x,v)}_{\mathcal{M}_{\mathsf{StrongCOH}}} \langle M, T'\rangle}$$

That is, when a thread reads from a message, it does not update its view by the message's view but just by its timestamp.[3] This change makes the semantics weaker: StrongCOH allows weak behavior of MP and SB+RMWs from Examples 2.17 and 2.19.

We include the same silent memory transitions in $\mathcal{M}_{\mathsf{StrongCOH}}$ as we do for $\mathcal{M}_{\mathsf{RA}}$, which is enough to guarantee termination of memory-fair executions of Rloop for the same reason as for RA.

[3]In this model one may change messages to not store views at all since they are never used. We keep the message views only in order to be as close as possible to RA.

3 PRELIMINARIES ON DECLARATIVE SEMANTICS

In this section, we review the declarative (a.k.a. axiomatic) framework for assigning semantics to concurrent programs and present the well-known declarative models for the four operational models presented above. Later, we will extend the framework and the existing correspondence results with fairness guarantees that account for infinite behaviors.

Relations. Given a binary relation (in particular, a function) R, $dom(R)$ and $codom(R)$ denote its domain and codomain. We write $R^?$, R^+, and R^* respectively to denote its reflexive, transitive, and reflexive-transitive closures. The inverse relation is denoted by R^{-1}. We denote by $R_1 ; R_2$ the (left) composition of two relations R_1, R_2, and assume that ; binds tighter than $\cup$ and $\setminus$. We denote by $[A]$ the identity relation on a set A. In particular, $[A] ; R ; [B] = R \cap (A \times B)$. For $n \geq 0$ and a relation R on a set A, R^n is recursively defined by $R^0 \triangleq [A]$ and $R^{n+1} \triangleq R ; R^n$. We write $R^{\leq n}$ for the union $\bigcup_{1 \leq i \leq n} R^n$.

Events. Events represent individual memory accesses in a run of a program. They consist of a thread identifier, an event label, and a serial number used to uniquely identify events and order the events inside each thread.

Definition 3.1. An *event* e is a tuple $\langle k, \tau : l \rangle$ where $k \in \mathbb{N} \cup \{\bot\}$ is a serial number inside each thread ($\bot$ for initialization events), $\tau \in \mathsf{Tid} \uplus \{\bot\}$ is a thread identifier ($\bot$ for initialization events), and $l \in \mathsf{ELab}$ is an event label (as defined in Def. 2.1). The functions $\mathtt{sn}$, $\mathtt{tid}$, and $\mathtt{elab}$ return the serial number, thread identifier, and the event label of an event. The functions $\mathtt{typ}$, $\mathtt{loc}$, $\mathtt{val}_\mathtt{r}$, and $\mathtt{val}_\mathtt{w}$ are lifted to events in the obvious way. We denote by Event the set of all events, and use $\mathtt{R}$, $\mathtt{W}$, and $\mathtt{RMW}$ to denote the following subsets:

$$\mathtt{R} \triangleq \{e \in \mathsf{Event} \mid \mathtt{typ}(e) = \mathtt{R} \vee \mathtt{typ}(e) = \mathtt{RMW}\}$$
$$\mathtt{W} \triangleq \{e \in \mathsf{Event} \mid \mathtt{typ}(e) = \mathtt{W} \vee \mathtt{typ}(e) = \mathtt{RMW}\}$$
$$\mathtt{RMW} \triangleq \{e \in \mathsf{Event} \mid \mathtt{typ}(e) = \mathtt{RMW}\}$$

We use subscripts and superscripts to restrict sets of events to certain location and thread (*e.g.,* $\mathtt{W}_x = \{w \in \mathtt{W} \mid \mathtt{loc}(w) = x\}$ and $E^\tau = \{e \in E \mid \mathtt{tid}(e) = \tau\}$). The set of *initialization events* is given by $\mathsf{Init} \triangleq \{\langle \bot, \bot : \mathtt{W}(x, 0) \rangle \mid x \in \mathsf{Loc}\}$.

Notation 3.2. *We denote by* $R|_{\mathtt{loc}}$ *the restriction of a relation* R *to events of the same location:*

$$R|_{\mathtt{loc}} = \{\langle e_1, e_2 \rangle \in R \mid \exists x \in \mathsf{Loc}.\ \mathtt{loc}(e_1) = \mathtt{loc}(e_2) = x\}$$

Our representation of events induces a *sequenced-before* partial order on events given by:

$$e_1 < e_2 \overset{\triangle}{\Leftrightarrow} (e_1 \in \mathsf{Init} \wedge e_2 \notin \mathsf{Init}) \vee (\mathtt{tid}(e_1) = \mathtt{tid}(e_2) \wedge \mathsf{sn}(e_1) < \mathsf{sn}(e_2))$$

Initialization events precede all non-initialization events, while events of the same thread are ordered according to their serial numbers.

Behaviors (*i.e.,* mappings from threads to sequential traces) are associated with sets of events in the obvious way:

Definition 3.3. The set of events *extracted from a behavior* β, denoted by $\mathsf{Event}(\beta)$, is given by $\mathsf{Event}(\beta) \triangleq \mathsf{Init} \cup \{\langle k, \tau : \beta(\tau)(k) \rangle \mid \tau \in \mathsf{Tid}, k \in dom(\beta(\tau))\}$.

It is easy to see that for every behavior β, $\mathsf{Event}(\beta)$ satisfies certain "well-formedness" properties:

Definition 3.4. A set $E \subseteq \mathsf{Event}$ is *well-formed* if the following hold:

- $\mathsf{Init} \subseteq E$.
- $\mathtt{tid}(e) \neq \bot$ and $\mathsf{sn}(e) \neq \bot$ for every $e \in E \setminus \mathsf{Init}$.

- If $\mathtt{tid}(e_1) = \mathtt{tid}(e_2)$ and $\mathtt{sn}(e_1) = \mathtt{sn}(e_2)$, then $e_1 = e_2$ for all $e_1, e_2 \notin \mathsf{Init}$.
- For every $e \in E \setminus \mathsf{Init}$ and $0 \leq k < \mathtt{sn}(e)$, there exists $l \in \mathsf{ELab}$ such that $\langle k, \mathtt{tid}(e) : l\rangle \in E$.

Execution Graphs. An execution graph consists of a set of events, a *reads-from* mapping that determines the write event from which each read reads its value, and a *modification order* which totally orders the writes to each location.

Definition 3.5. An *execution graph* G is a tuple $\langle E, rf, mo\rangle$ where:

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

We denote the components of G by $G.\mathsf{E}$, $G.\mathtt{rf}$, and $G.\mathtt{mo}$, and write $G.\mathsf{po}$ (called *program order*) for the restriction of sequenced-before to $G.\mathsf{E}$ (*i.e.,* $G.\mathsf{po} \triangleq [G.\mathsf{E}]; <; [G.\mathsf{E}]$). For a set $E' \subseteq \mathsf{Event}$, we write $G.E'$ for $G.\mathsf{E} \cap E'$ (*e.g.,* $G.\mathtt{W} = G.\mathsf{E} \cap \mathtt{W}$). The set of all execution graphs is denoted by EGraph.

A *declarative memory system* is simply a set $\mathcal{G}$ of execution graphs (often formulated using a conjunction of several constraints). We refer to execution graphs in a declarative memory system $\mathcal{G}$ as $\mathcal{G}$*-consistent* execution graphs.

We can now define the behaviors allowed by a given declarative memory system.

Definition 3.6. A behavior β is *allowed by a declarative memory system* $\mathcal{G}$ if $\mathsf{Event}(\beta) = G.\mathsf{E}$ for some execution graph $G \in \mathcal{G}$. We denote by $\mathsf{B}(\mathcal{G})$ ($\mathsf{B}^{\mathsf{fin}}(\mathcal{G})$) the set of all (finite) behaviors that are allowed by $\mathcal{G}$.

The linking with programs is defined as follows.

Definition 3.7. Let P be a program, $\mathcal{G}$ be a declarative memory system, and β be a behavior.

- β is a *behavior of* P *under* $\mathcal{G}$ if $\beta \in \mathsf{B}(P) \cap \mathsf{B}(\mathcal{G})$.
- β is a *thread-fair behavior of* P *under* $\mathcal{G}$ if $\beta \in \mathsf{B}^{\mathsf{tf}}(P) \cap \mathsf{B}(\mathcal{G})$.

3.1 A Declarative Memory System for SC

To provide a declarative formulation of SC, following Alglave et al. [2014], we use the standard "from-read" relation (a.k.a. "reads-before"). In this relation a read r is ordered before a write w if r reads from a write w' that is earlier than w in the modification order.

Definition 3.8. The *from-read* relation for an execution graph G, denoted by $G.\mathtt{fr}$, is defined by:

$$G.\mathtt{fr} \triangleq (G.\mathtt{rf}^{-1}\,;\, G.\mathtt{mo}) \setminus [G.\mathsf{E}].$$

Note that we have to explicitly subtract the identity relation from $G.\mathtt{rf}^{-1}\,;\,G.\mathtt{mo}$ for making sure that $\mathtt{RMW}$ events are not $G.\mathtt{fr}$-ordered before themselves.

Having defined $\mathtt{fr}$, the "SC-happens-before" relation is given by:

$$G.\mathsf{hb}_{\mathsf{SC}} \triangleq (G.\mathsf{po} \cup G.\mathtt{rf} \cup G.\mathtt{mo} \cup G.\mathtt{fr})^{+}$$

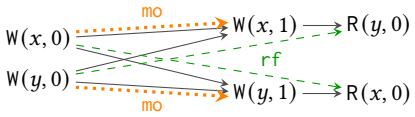
In turn, SC consistency requires that $G.\mathsf{hb}_{\mathsf{SC}}$ is irreflexive:

$$\mathcal{G}_{\mathsf{SC}} \triangleq \{G \in \mathsf{EGraph} \mid G.\mathsf{hb}_{\mathsf{SC}} \text{ is irreflexive}\}$$

Intuitively speaking, every trace of $\mathcal{M}_{\mathsf{SC}}$ induces an execution graph G with irreflexive $G.\mathsf{hb}_{\mathsf{SC}}$; and, conversely, every total order on $G.\mathsf{E}$ that extends $G.\mathsf{hb}_{\mathsf{SC}}$ is essentially a trace of $\mathcal{M}_{\mathsf{SC}}$. The following standard theorem formalizes these claims for *finite* executions:

Theorem 3.9 ([Alglave et al. 2014]). $\mathsf{B}^{\mathsf{fin}}(\mathcal{M}_{\mathsf{SC}}) = \mathsf{B}^{\mathsf{fin}}(\mathcal{G}_{\mathsf{SC}})$.

Example 3.10. $\mathcal{G}_{\mathsf{SC}}$ forbids the annotated outcome of the SB program from Example 2.15 because the following graph is $\mathcal{G}_{\mathsf{SC}}$-inconsistent ($\mathtt{W}(x,0)$ and $\mathtt{W}(y,0)$ are the implicit initialization writes):



Indeed, to get the desired behavior, the rf-edges are forced because of the read values. Since mo cannot contradict po (they are both included in $\mathsf{hb}_{\mathsf{SC}}$), the mo-edges are also forced as depicted above. We obtain fr-edges from $\mathtt{R}(x,0)$ to $\mathtt{W}(x,1)$ and from $\mathtt{R}(y,0)$ to $\mathtt{W}(y,1)$, which, in turn, imply a $\mathsf{hb}_{\mathsf{SC}}$-cycle composed of two po and two fr edges.

3.2 A Declarative Memory System for TSO

Following Alglave et al. [2014], a declarative formulation for TSO is easily obtained from the one of SC, by removing from the transitive closure in $\mathsf{hb}_{\mathsf{SC}}$ the program order edges from writes to reads that are not necessarily "preserved" in TSO. Indeed, because writes are buffered in TSO, roughly speaking, the effect of a write in TSO may be delayed w.r.t. subsequent reads. By contrast, it cannot be delayed w.r.t. subsequent writes, since entries in the TSO buffers propagate in a FIFO fashion.

When removing the write to read program order edges, we need to explicitly enforce "SC per-location" (a.k.a. coherence), which takes care of intra-thread write-read pairs (a read r from x that is later in program order than a write w to x may not read from a write that is mo-earlier than w). To achieve this, the model employs the following derived relations:

$$G.\mathtt{rfe} \triangleq G.\mathtt{rf} \setminus G.\mathtt{po} \qquad \textit{(external reads-from)}$$

$$G.\mathtt{ppo} \triangleq G.\mathtt{po} \setminus ((\mathtt{W} \setminus \mathtt{RMW}) \times (\mathtt{R} \setminus \mathtt{RMW})) \qquad \textit{(preserved program order)}$$

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

$$G.\mathtt{sc}_{\mathtt{loc}} \triangleq (G.\mathtt{po}|_{\mathtt{loc}} \cup G.\mathtt{rf} \cup G.\mathtt{mo} \cup G.\mathtt{fr})^{+} \qquad \textit{(SC-per-location order)}$$

Then, TSO consistency requires that $G.\mathsf{hb}_{\mathsf{TSO}}$ and $G.\mathtt{sc}_{\mathtt{loc}}$ are irreflexive:

$$\mathcal{G}_{\mathsf{TSO}} \triangleq \{G \in \mathsf{EGraph} \mid G.\mathsf{hb}_{\mathsf{TSO}} \text{ and } G.\mathtt{sc}_{\mathtt{loc}} \text{ are irreflexive}\}$$

Theorem 3.11 ([Alglave et al. 2014]). $\mathsf{B}^{\mathsf{fin}}(\mathcal{M}_{\mathsf{TSO}}) = \mathsf{B}^{\mathsf{fin}}(\mathcal{G}_{\mathsf{TSO}})$.

The execution graph for the SB program in Example 3.10 is $\mathcal{G}_{\mathsf{TSO}}$-consistent. In particular, the two po edges that participate in the $G.\mathsf{hb}_{\mathsf{SC}}$ cycle are from a write to a read, so none of them is included in $G.\mathsf{hb}_{\mathsf{TSO}}$.

3.3 A Declarative Memory System for RA

The declarative model for RA is obtained by strengthening the SC per-location requirement to use RA's happens-before relation instead of the program order:

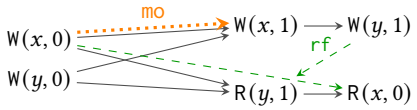
$$G.\mathsf{hb}_{\mathsf{RA}} \triangleq (G.\mathtt{po} \cup G.\mathtt{rf})^{+} \qquad \textit{(RA-happens-before)}$$

$$G.\mathtt{ra}_{\mathtt{loc}} \triangleq (G.\mathsf{hb}_{\mathsf{RA}}|_{\mathtt{loc}} \cup G.\mathtt{rf} \cup G.\mathtt{mo} \cup G.\mathtt{fr})^{+} \qquad \textit{(RA-per-location order)}$$

Then, RA consistency requires that $G.\mathtt{ra}_{\mathtt{loc}}$ is irreflexive:

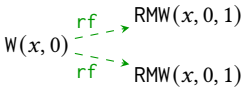
$$\mathcal{G}_{\mathsf{RA}} \triangleq \{G \in \mathsf{EGraph} \mid G.\mathtt{ra}_{\mathtt{loc}} \text{ is irreflexive}\}$$

Example 3.12. The annotated outcome of the MP program from Example 2.17 is disallowed by $\mathcal{G}_{\mathsf{RA}}$ because the following (partially depicted) execution graph is $\mathcal{G}_{\mathsf{RA}}$-inconsistent:



An execution graph for this outcome must have rf and mo-edges as depicted above. Since mo goes from $\mathtt{W}(x,0)$ to $\mathtt{W}(x,1)$, and $\mathtt{R}(x,0)$ reads from $\mathtt{W}(x,0)$, we have an fr edge from $\mathtt{R}(x,0)$ to $\mathtt{W}(x,1)$. Due to the $\mathsf{hb}_{\mathsf{RA}}$ from $\mathtt{W}(x,1)$ to $\mathtt{R}(x,0)$, we obtain a $\mathtt{ra}_{\mathtt{loc}}$-cycle, rendering this graph $\mathcal{G}_{\mathsf{RA}}$-inconsistent.

Example 3.13. Similarly, the annotated outcome of 2RMW from Example 2.18 is disallowed by $\mathcal{G}_{\mathsf{RA}}$ because the following execution graph is $\mathcal{G}_{\mathsf{RA}}$-inconsistent for any choice of mo:



To see this, note that in $\mathcal{G}_{\mathsf{RA}}$-consistent executions, mo cannot contradict po. Hence, we must have mo from the initial write to the two RMWs. This implies an fr edge in both directions between the two RMWs, so that $\mathtt{ra}_{\mathtt{loc}}$ must be cyclic.

Equivalence to the operational RA model for *finite* behaviors follows from [Kang et al. 2017]:

Theorem 3.14. $\mathsf{B}^{\mathsf{fin}}(\mathcal{M}_{\mathsf{RA}}) = \mathsf{B}^{\mathsf{fin}}(\mathcal{G}_{\mathsf{RA}})$.

3.4 A Declarative Memory System for StrongCOH

The declarative model for StrongCOH is obtained by requiring "SC per-location" and irreflexivity of RA's happens-before, $(G.\mathsf{po} \cup G.\mathtt{rf})^+$:

$$\mathcal{G}_{\mathsf{StrongCOH}} \triangleq \{G \in \mathsf{EGraph} \mid G.\mathsf{hb}_{\mathsf{RA}} \text{ and } G.\mathsf{sc}_{\mathtt{loc}} \text{ are irreflexive}\}$$

Similarly to RA, equivalence to the operational StrongCOH model for *finite* behaviors follows from the results of Kang et al. [2017]:

Theorem 3.15. $\mathsf{B}^{\mathsf{fin}}(\mathcal{M}_{\mathsf{StrongCOH}}) = \mathsf{B}^{\mathsf{fin}}(\mathcal{G}_{\mathsf{StrongCOH}})$.

4 MAKING DECLARATIVE SEMANTICS FAIR

In this section, we introduce memory fairness into declarative models in a model-agnostic fashion.

To define fairness of execution graphs, we require that the partial ordering of events in the graph is, like the ordering of natural numbers, *prefix-finite*. From an operational point of view, an event preceded by an infinite number of events is never executed.

Definition 4.1. A relation R on a set A is *prefix-finite* if $\{a \mid \langle a, b\rangle \in R\}$ is finite for every $b \in A$.

Concretely, we require the modification order and the from-read relation to be prefix-finite.[4]

Definition 4.2. An execution graph G is *fair* if $G.\mathtt{mo}$ and $G.\mathtt{fr}$ are prefix-finite. We denote by $\mathcal{G}^{\mathtt{fair}}$ the set of all fair execution graphs, and let $\mathcal{G}_X^{\mathtt{fair}} \triangleq \mathcal{G}_X \cap \mathcal{G}^{\mathtt{fair}}$ for $X \in \{\mathsf{SC}, \mathsf{TSO}, \mathsf{RA}, \mathsf{StrongCOH}\}$.

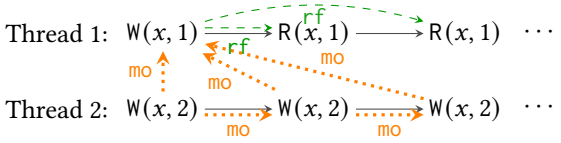
[4]Note that the *program order* and the *reads-from* relation are prefix-finite in a well-formed execution graph. The former–by construction, the latter–since its reverse relation is functional.

Example 4.3. The following program illustrates our definition of fairness:

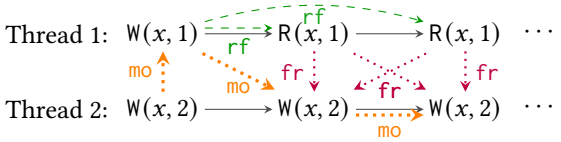
$$\begin{array}{l} x := 1\,; \\ L_1 \colon a := x \;\textit{// only 1} \\ \textbf{goto}\ L1 \end{array} \;\Big\|\; \begin{array}{l} L_2 \colon x := 2\,; \\ \quad \textbf{goto}\ L_2 \end{array} \qquad \text{(SCDeclUnfair)}$$

Thread-fair executions of this program cannot produce the annotated outcome with the SC memory system. With the declarative SC memory system, however, there are two ways in which every read can read from the write of 1.

First, the write of 1 to x may have infinitely many mo-predecessors, as illustrated below.



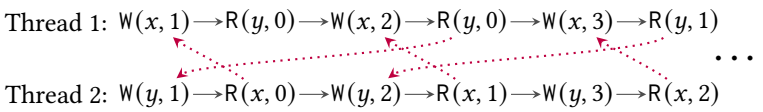
Otherwise, the write of 1 may have finitely many mo-predecessors but infinitely many mo-successors. Then, each of the mo-successors will have infinitely many fr-predecessors.



In both cases, the execution graph is unfair. (As we prove below, this is not a coincidence.)

Example 4.4. On the converse, one should avoid unnecessary prefix-finiteness constraints. In particular, requiring prefix-finiteness of cyclic relations, such as $[G.\mathsf{E} \setminus \mathsf{Init}]\,;\mathsf{hb}_{\mathsf{SC}}$ under TSO, RA, or StrongCOH, is too strong. Doing so would forbid the annotated behavior of the following example. The corresponding execution graph contains an infinite $\mathsf{po} \cup \mathsf{fr}$ descending chain. Yet, the three models allow the annotated behavior, as every write may be delayed past 1 or 2 reads.

$$\begin{array}{l} L_1 \colon k := k + 1 \\ \quad x := k \\ \quad a := y \;\textit{// 0,0,1,2...} \\ \quad \textbf{goto}\ L_1 \end{array} \;\Big\|\; \begin{array}{l} L_2 \colon m := m + 1 \\ \quad y := m \\ \quad b := x \;\textit{// 0,1,2,3...} \\ \quad \textbf{goto}\ L_2 \end{array} \qquad \text{(HbAcyclic)}$$



Our main result extends Theorems 3.9, 3.11, 3.14 and 3.15 for *infinite* traces by imposing memory fairness on the operational systems (Def. 2.12) and execution graph fairness on the declarative systems (Def. 4.2).

Theorem 4.5. *For* $X \in \{\mathsf{SC}, \mathsf{TSO}, \mathsf{RA}, \mathsf{StrongCOH}\}$,

$$\mathsf{B}^{\mathsf{mf}}(\mathcal{M}_X) = \mathsf{B}(\mathcal{G}_X^{\mathsf{fair}}).$$

As a corollary, it easily follows from our definitions that the set of (thread&) memory-fair behaviors of a program P under $\mathcal{M}_X$ coincides with the set of (thread&) memory-fair behaviors of a program P under $\mathcal{G}_X^{\mathsf{fair}}$.

The full proof of Theorem 4.5 is included in appendix ([Lahav et al. 2021a]) and its Coq mechanization in [Lahav et al. 2021b]. Here, we outline the proof starting with the easier direction.

4.1 $\mathsf{B}^{\mathsf{mf}}(\mathcal{M}_X) \subseteq \mathsf{B}(\mathcal{G}_X^{\mathtt{fair}})$

Given a memory-fair behavior β of $\mathcal{M}_X$, we let ρ be a memory-fair observable trace of $\mathcal{M}_X$ such that $\beta(\rho) = \beta$. Then, using ρ, we construct a fair execution graph $G \in \mathcal{G}_X$. Its events are determined by β ($G.\mathsf{E} = \mathsf{Event}(\beta)$), and its relations are defined differently for every system:

SC. The rf and mo relations are determined by the trace order: for each read rf assigns the latest write of the same location, while mo corresponds to the trace order restricted to writes to the same location. It follows that fr is included in the trace order, and since the trace order is prefix-finite, mo and fr are prefix-finite as well.

TSO. We define mo to be the order in which writes to the same location are propagated to memory. For each read, rf maps it either to the mo-maximal write to the same location that was propagated before it in ρ (if the read reads from memory) or to the po-maximal one by the same thread (if it reads from the buffer). Since every write is eventually propagated to memory, and once propagated no thread can read from an mo-prior write, it follows that both mo and fr are prefix-finite.

RA and StrongCOH. The mo component of G follows the order induced by timestamps of messages in the operational run. Prefix-finiteness of mo follows from the facts that a location and a timestamp uniquely identify the corresponding message (and the write event in G respectively) and that timestamps are natural numbers—that is, each write event w representing a message with a timestamp t has at most t mo-prior writes.

The rf component of G connects an event related to a read/RMW transition of ρ with a write event representing the message read by the transition.

Prefix-finiteness of fr follows from the fact that in the fair operational run every message is eventually propagated to every thread. That is, for any given write event w to a location x in G representing a message with a timestamp t, there cannot be infinitely many reads from x in G reading from write events that correspond to messages with timestamps smaller than t.

4.2 $\mathsf{B}(\mathcal{G}_X^{\mathtt{fair}}) \subseteq \mathsf{B}^{\mathsf{mf}}(\mathcal{M}_X)$

The converse direction is more challenging. Given a fair $\mathcal{G}_X$-consistent execution graph G, we have to find a memory-fair observable trace ρ of $\mathcal{M}_X$ such that $\mathsf{Event}(\beta(\rho)) = G.\mathsf{E}$.

Put differently, we need a total order over $G.\mathsf{E} \setminus \mathsf{Init}$ that extends $G.\mathsf{po}$, so that some memory-fair run of $\mathcal{M}_X$ executes according to this order. Existing proofs of correspondence between declarative and operational definitions of SC, RA, and StrongCOH pick an arbitrary total order extending $G.\mathsf{hb}_{\mathsf{SC}}$ (for SC) and $G.\mathsf{hb}_{\mathsf{RA}}$ (for RA and StrongCOH). (Assuming the axiom of choice, any partial order R on a set A can be extended to a total order on A.) It is then not difficult to show that executing the program following that order yields the labels appearing in the execution graph. For infinite graphs, however, an arbitrary extension of $G.\mathsf{hb}_{\mathsf{SC}}$ (or $G.\mathsf{hb}_{\mathsf{RA}}$ respectively) does not necessarily correspond to a (memory-fair) run of the program. For this, we need an *enumeration* of $G.\mathsf{E} \setminus \mathsf{Init}$, as defined next.

Definition 4.6. An *enumeration* of a set A is a (finite or infinite) injective (*i.e.,* without repetitions) sequence ν covering all the elements in A (*i.e.,* $A = \{\nu(i) \mid i \in dom(\nu)\}$). An enumeration ν of A *respects* a partial order R on A if $i < j$ whenever $\langle \nu(i), \nu(j) \rangle \in R$.

Prefix-finiteness of a partial order ensures that a suitable enumeration exists (our proof employs classical, non-constructive, reasoning):

Proposition 4.7. *Let R be a prefix-finite partial order on a countable set A. Then, there exists an enumeration of A that respects R.*

However, we do not yet have that the "happens-before" relation of each model is prefix-finite; we only know that $G.\mathsf{mo}$ and $G.\mathsf{fr}$ are prefix-finite. Next, we show that prefix-finiteness of $G.\mathsf{mo}$ and $G.\mathsf{fr}$ suffices for prefix-finiteness of the other relations, as long as the program in question has a bounded number of threads. (Recall that we assume that the set Tid is finite.)

First, note that every relation on a finite set is prefix-finite, and prefix-finiteness is preserved by (finite) composition.

Lemma 4.8. *Let R and R' be prefix-finite relations and $n \in \mathbb{N}$. Then $R \cup R'$, $R\,;R'$ and $R^{\leq n}$ are also prefix-finite.*

For transitive closures, we need an auxiliary property.

Definition 4.9. A relation R on a set A is *n-total* if for every $n+1$ distinct elements $a_1, \ldots, a_{n+1} \in A$, we have $\langle a_i, a_j \rangle \in R$ for some $1 \leq i, j \leq n+1$.

For an execution graph G with n threads, $G.\mathsf{po}$ is n-total (as a relation on $G.\mathsf{E} \setminus \mathsf{Init}$). By the pigeonhole principle, any set of $n+1$ events in $G.\mathsf{E} \setminus \mathsf{Init}$ contain two elements belonging to the same thread, and those two events are ordered by $G.\mathsf{po}$.

Now, if a relation R is n-total and acyclic, its transitive closure R^+ has bounded length, which entails that R^+ is prefix-finite provided R is prefix-finite.

Lemma 4.10. *Let R be an acyclic, n-total, prefix-finite relation. Then, R^+ is prefix-finite.*

As a corollary, we obtain that the prefix-finiteness of the "happens-before" relation in fair execution graphs.

Corollary 4.11. *For $X \in \{\mathsf{SC}, \mathsf{TSO}, \mathsf{RA}, \mathsf{StrongCOH}\}$, let G be a fair $\mathcal{G}_X$-consistent execution graph. Then $[G.\mathsf{E} \setminus \mathsf{Init}]\,;G.\mathsf{hb}_X$ is prefix finite.*[5]

From Prop. 4.7, there is an enumeration ν that respects hb_X. We use ν to construct a program trace ρ:

SC. The trace ρ follows ν exactly. Since $\mathcal{M}_{\mathsf{SC}}$ has no silent memory transitions, ρ is trivially memory fair.

TSO. The trace ρ is incrementally constructed by following the order of events in ν and appending an appropriate sequence of transitions. If the next event in ν is a read, we append to ρ all unexecuted po-prior writes and then the read. If the next event in ν is a write, we append it to the trace if it has not already been included in the trace. In addition, when the next event is a write, we append its propagation action. By construction, every write in ρ is eventually propagated to memory.

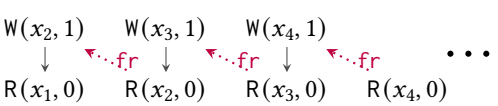
RA and StrongCOH. The trace ρ is the enumeration ν interleaved with silent RA/StrongCOH transition labels. Namely, for each write w and thread τ, we compute an index i in the enumeration such that it is *safe* to propagate w to τ at that index: for each event in τ with index greater than i, there is no X-following (where $\mathsf{X} = \mathsf{hb}_{\mathsf{RA}}$ for RA and $\mathsf{X} = \mathsf{rf}^?\,;\mathsf{po}^?$ for StrongCOH) (i) write that mo-precedes w and (ii) read that reads from a write mo-preceding w. Since G is fair, such an index is defined for all (non-initialization) writes. Then, after the event with an index corresponding to some write has been enumerated, we execute a propagation transition for the write. In that way, every write is eventually propagated to every thread, so the resulting trace is memory fair.

Remark 3. Corollary 4.11 relies on having a bounded number of threads. With infinite number of threads, generated, *e.g.*, by thread spawning, prefix-finiteness of mo and fr is not enough to rule

[5] We define $G.\mathsf{hb}_{\mathsf{StrongCOH}}$ to be equal to $G.\mathsf{hb}_{\mathsf{RA}}$.

out unfair behaviors. To see this, consider the annotated behavior of the following program and the corresponding execution graph:

$$L\colon i := i + 1$$
$$\textbf{spawn} \left\{ \begin{array}{l} x_{i+1} := 1 \\ a := x_i \ \textit{// only 0} \end{array} \right\}$$
$$\textbf{goto}\ L$$



While mo and fr are trivially prefix-finite, $\mathsf{hb}_{\mathsf{SC}}$ has an infinite descending chain, and indeed there is no SC execution of the program leading to the annotated behavior (where spawn adds a thread to the current pool, and a thread from the pool is non-deterministically chosen at each step).

4.3 Making RC11 Fair

Having established evidence for the adequacy of the declarative fairness condition, we may apply this condition in other (and richer) declarative models. In particular, we propose to adopt this condition into the C/C++ memory model. Next, we discuss this proposal in the context of the RC11 model [Lahav et al. 2017], a repaired version of the C/C++11 specification [Batty et al. 2012] that fixes certain issues involving sequentially consistent accesses and works around the "thin-air" problem by completely forbidding $\mathsf{po} \cup \mathsf{rf}$ cycles. A full definition of RC11 is obtained by carefully combining the key concepts of SC, RA, and StrongCOH. It requires us to include in the declarative framework access modes (a.k.a. "memory orderings"—the consistency level required from every memory access), and several types of fences. For simplicity, we elide these definitions and keep the discussion more abstract. Indeed, there is nothing special about RC11 in this context—the declarative fairness condition could be added to any model requiring $\mathsf{po} \cup \mathsf{rf}$ acyclicity.

Generally speaking, when proposing a strengthening of a programming language memory model, one has to make sure that the mapping schemes to multicore architectures are not broken, and that source-to-source compiler transformations are still validated. In our case, the mapping of RC11 to x86-TSO trivially remains sound. Indeed, as we saw in Theorem 4.5, the natural operational characterization of liveness in TSO corresponds to the declarative condition requiring that the mo and fr relations are prefix-finite. Since the same condition is applied both in the source level (RC11) and in the target level (x86-TSO), and mappings of source graphs to target ones keep mo and fr intact, we maintain the soundness of the known mappings.[6] We note that for establishing the soundness of the mappings to other architectures, one first needs a formal fairness condition of the architecture. While this may be more difficult in architectures weaker than x86-TSO (see §6), it is likely that no hardware will allow that a write is placed after infinitely many other writes in the coherence order (non-prefix-finite mo), or that infinitely many reads do not observe a later write (non-prefix-finite fr).

Considering compiler transformations, one has to show that every behavior of the target program explained by a consistent graph G_{tgt} is also obtained by a consistent graph G_{src} of the source program. It is not hard to see that the constructions of Vafeiadis et al. [2015] and Lahav et al. [2017] work as-is for the RC11 model strengthened with fairness. First, the constructions of G_{src} for *reordering transformations*, which reorder two memory accesses under certain conditions, keep the same mo and fr relations of G_{tgt}; so their prefix-finiteness trivially follows.

Second, we consider *elimination transformations* that eliminate a redundant memory access. In this case, G_{src} is obtained from G_{tgt} by adding one additional event e_{new} that corresponds to the eliminated instruction. For read elimination (read-after-read or read-after-write), e_{new} is a read

[6] See http://www.cl.cam.ac.uk/~pes20/cpp/cpp0xmappings.html [accessed July-2021].

event, and the construction of ensures that $G_{\text{src}}.\mathtt{mo} = G_{\text{tgt}}.\mathtt{mo}$. In turn, $G_{\text{tgt}}.\mathtt{fr} \subseteq G_{\text{src}}.\mathtt{fr}$, but since only one event is added to G_{src}, prefix-finiteness of $\mathtt{fr}$ is again trivially preserved.

Finally, we consider write-after-write elimination. Let w_0 denote the immediate $G_{\text{src}}.\mathsf{po}$-successor of e_{new}. Then, to construct $G_{\text{src}}.\mathtt{mo}$, one places e_{new} as the immediate predecessor of w_0. Then, consistency of G_{src} follows the argument of [Lahav et al. 2017], and it remains to show that fairness of G_{src} follows from the fairness of G_{tgt}. The latter is easy: write events in G_{src} other than e_{new} all have at most one more incoming $G_{\text{src}}.\mathtt{mo}$ edge (from e_{new}), and the same set of incoming $G_{\text{src}}.\mathtt{fr}$ edges. In turn, For e_{new} itself, we have: $\{e \in G_{\text{src}}.\mathsf{E} \mid \langle e, e_{\text{new}}\rangle \in G_{\text{src}}.\mathtt{mo} \cup G_{\text{src}}.\mathtt{fr}\} \subseteq \{e \in G_{\text{tgt}}.\mathsf{E} \mid \langle e, w_0\rangle \in G_{\text{tgt}}.\mathtt{mo} \cup G_{\text{tgt}}.\mathtt{fr}\}$.

In the next section, we demonstrate that adding fairness to RC11 as proposed above provides the necessary underpinnings allowing one to formally reason about termination under RC11.

4.4 From Finite to Infinite Robustness

Common advice given to programmers of multi-threaded software is to follow a programming discipline that hides the effects of that weak memory model, *e.g.,* to use exclusively sequentially consistent accesses. Programs that follow such a discipline are *robust*, meaning that they have only sequentially consistent behaviors on the underlying weak memory model. While there is a rich literature on programming disciplines that imply robustness and verification techniques for robustness [Bouajjani et al. 2013, 2018, 2011; Derevenetc and Meyer 2014; Lahav and Margalit 2019; Margalit and Lahav 2021; Oberhauser 2018], most work only considers *finite* behaviors, i.e., they leave open whether programs following the discipline have only sequentially consistent *infinite* behaviors. This means that any correctness properties that only concern infinite behaviors, such as starvation-freedom, might be lost on the weak memory model despite its (finite) robustness. In this section, we show that this cannot happen as long as the weak memory model satisfies our declarative memory fairness condition and its consistency predicate is $\mathsf{po} \cup \mathtt{rf}$-prefix closed. This is the case for all models studied in this paper. Thus our unified definition of memory fairness lifts all existing robustness results for these models from the literature to infinite behaviors.

First, we observe that the consistency predicates based on acyclicity (SC-consisterncy, in particular) enjoy a "compactness property"—if they hold for all finite prefixes of a graph, then they also hold for the full graph. Below, by *finite* execution graph, we mean a graph G with $G.\mathsf{E} \setminus \mathsf{Init}$ being finite (the set Init of initialization events may be infinite if Loc is infinite).

Definition 4.12. An execution graph G' is a $\mathsf{po} \cup \mathtt{rf}$*-prefix* of an execution graph G if we have $dom((G.\mathsf{po} \cup G.\mathtt{rf})\,;\,[G'.\mathsf{E}]) \subseteq G'.\mathsf{E}$, $G'.\mathtt{rf} = [G'.\mathsf{E}];G.\mathtt{rf};[G'.\mathsf{E}]$, and $G'.\mathtt{mo} = [G'.\mathsf{E}];G.\mathtt{mo};[G'.\mathsf{E}]$.

Proposition 4.13 ($\mathcal{G}_{\mathsf{SC}}$ compactness). *Let G be an execution graph with prefix-finite $([G.\mathsf{E} \setminus \mathsf{Init}]\,;\,G.\mathsf{po} \cup G.\mathtt{rf})^+$. If every finite $\mathsf{po} \cup \mathtt{rf}$-prefix of G is $\mathcal{G}_{\mathsf{SC}}$-consistent, then so is G.*

Proof. Suppose that G is $\mathcal{G}_{\mathsf{SC}}$-inconsistent, and let $a_1, \ldots, a_n \in G.\mathsf{E}$ such that $\langle a_i, a_{i+1}\rangle \in G.\mathsf{po} \cup G.\mathtt{rf} \cup G.\mathtt{mo} \cup G.\mathtt{fr}$ for every $1 \le i \le n-1$, and $\langle a_n, a_1\rangle \in G.\mathsf{po} \cup G.\mathtt{rf} \cup G.\mathtt{mo} \cup G.\mathtt{fr}$. Let $E' = \mathsf{Init} \cup dom((G.\mathsf{po} \cup G.\mathtt{rf})^*\,;\,[\{a_1, \ldots, a_n\}])$, and let $G' = \langle E', [E']\,;\,G.\mathtt{rf}\,;\,[E'], [E']\,;\,G.\mathtt{mo}\,;\,[E']\rangle$. Since $([G.\mathsf{E} \setminus \mathsf{Init}]\,;\,G.\mathsf{po} \cup G.\mathtt{rf})^+$ is prefix-finite, G' is a finite $\mathsf{po} \cup \mathtt{rf}$-prefix of G. However, we have $\langle a_i, a_{i+1}\rangle \in G'.\mathsf{po} \cup G'.\mathtt{rf} \cup G'.\mathtt{mo} \cup G'.\mathtt{fr}$ for every $1 \le i \le n-1$, and $\langle a_n, a_1\rangle \in G'.\mathsf{po} \cup G'.\mathtt{rf} \cup G'.\mathtt{mo} \cup G'.\mathtt{fr}$, so G' is $\mathcal{G}_{\mathsf{SC}}$-inconsistent. □

Definition 4.14 (Robustness). Let P be a program and $\mathcal{G}$ be a declarative memory system.

- P is *finitely execution-graph robust* against $\mathcal{G}$ if for every finite behavior $\beta \in \mathsf{B}(P)$ and $G \in \mathcal{G}$ with $\mathsf{Event}(\beta) = G.\mathsf{E}$, we have $G \in \mathcal{G}_{\mathsf{SC}}$.
- P is *strongly execution-graph robust* against $\mathcal{G}$ if for every (finite or infinite) behavior $\beta \in \mathsf{B}(P)$ and $G \in \mathcal{G}$ with $\mathsf{Event}(\beta) = G.\mathsf{E}$, we have $G \in \mathcal{G}_{\mathsf{SC}}$.

Theorem 4.15. *Let $\mathcal{G}$ be a declarative memory system such that:*

- *$\mathcal{G}$-consistency is* $\mathsf{po} \cup \mathtt{rf}$*-prefix closed* (i.e., *if $G \in \mathcal{G}$ then $G' \in \mathcal{G}$ for every* $\mathsf{po} \cup \mathtt{rf}$*-prefix G' of G).*
- *$G \in \mathcal{G}$ implies that* $([G.\mathsf{E} \setminus \mathsf{Init}] ; G.\mathsf{po} \cup G.\mathtt{rf})^+$ *is prefix-finite.*

Then, if a program P is finitely execution-graph robust against $\mathcal{G}$, then it is also strongly execution-graph robust against $\mathcal{G}$.

Proof. Suppose that P is finitely execution-graph robust against $\mathcal{G}$. Let $G \in \mathcal{G}$ such that $G.\mathsf{E} = \mathsf{Event}(\beta)$ for some behavior $\beta \in \mathsf{B}(P)$. From finite execution-graph robustness, it follows that every finite $\mathsf{po} \cup \mathtt{rf}$-prefix of G is $\mathcal{G}_{\mathsf{SC}}$-consistent. By Prop. 4.13, G is $\mathcal{G}_{\mathsf{SC}}$-consistent as well. □

We note that the declarative TSO, RA, StrongCOH, and RC11 models satisfy the premises of Theorem 4.15. The Coq mechanization includes the formal proof of the statement below.

Corollary 4.16. *Suppose that a program P is finitely execution-graph robust against $\mathcal{G}_X$ for $X \in \{\mathsf{TSO}, \mathsf{RA}, \mathsf{StrongCOH}, \mathsf{RC11}\}$. Then, the set of (thread&) memory-fair behaviors of P under $\mathcal{M}_X$ coincides with the set of (thread&) memory-fair behaviors of P under $\mathcal{M}_{\mathsf{SC}}$.*

Proof. One direction is obvious since $\mathcal{M}_{\mathsf{SC}}$ is stronger than $\mathcal{M}_X$. For the converse, let β be a memory-fair behavior of P under $\mathcal{M}_X$. Then, by Theorem 4.5, we have that β be a memory-fair behavior of P under $\mathcal{G}_X^{\mathtt{fair}}$. By definition, we have that $\beta \in \mathsf{B}(P) \cap \mathsf{B}(\mathcal{G}_X^{\mathtt{fair}})$. Let $G \in \mathcal{G}_X$ such that $\mathsf{Event}(\beta) = G.\mathsf{E}$. Then, since $G \in \mathcal{G}_X$, by Theorem 4.15, we have that $G \in \mathcal{G}_{\mathsf{SC}}$. Since the declarative fairness condition is the same in all four models, we have $G \in \mathcal{G}_{\mathsf{SC}}^{\mathtt{fair}}$. Hence, we have $\beta \in \mathsf{B}(P) \cap \mathsf{B}(\mathcal{G}_{\mathsf{SC}}^{\mathtt{fair}})$, and so by Theorem 4.5, it follows that β is a memory-fair behavior of P under $\mathcal{M}_{\mathsf{SC}}$. To deal with thread fairness, one has to use $\mathsf{B}^{\mathsf{tf}}(P)$ instead of $\mathsf{B}(P)$ in this argument. □

As a simple application example, the SpinLock-Client program in §5.1 below is (finitely) execution-graph robust because the program employs only a single location (the location l for the lock implementation). Then, Corollary 4.16 entails that this program may diverge under the weak memory models studied in this paper iff it diverges under SC, and that the same also holds when assuming thread fairness.

5 PROVING DEADLOCK FREEDOM FOR LOCKS

In this section, we prove the termination and/or fairness of spinlock, ticket lock, and MCS lock clients. The key to doing so is Theorem 5.3 below, which reduces proving termination of spinloops under fair weak memory models to reasoning about a single specific iteration of the loop.

For simplicity, we henceforth assume that the sequential programs composing the concurrent programs are deterministic, as defined below. (The thread interleaving itself still makes the concurrent program semantics non-deterministic.)

Definition 5.1. A program P is *deterministic* if $\overline{p} \xrightarrow{\tau:l_1}_P \overline{p}_1$ and $\overline{p} \xrightarrow{\tau:l_2}_P \overline{p}_2$ imply that $\mathtt{typ}(l_1) = \mathtt{typ}(l_2)$ and $\mathtt{loc}(l_1) = \mathtt{loc}(l_2)$, and, moreover, if $l_1 = l_2$, then $\overline{p}_1 = \overline{p}_2$ also holds.

For a behavior β of a deterministic program P and $\tau \in \mathsf{Tid}$, we denote by $\mu_\tau(\beta)$ the unique run of $P(\tau)$ that induces the sequential trace $\beta(\tau)$.

Definition 5.2. A *spinloop iteration* of thread τ in a behavior β is a range of event serial numbers $[n, n']$ such that the sequence of corresponding program steps:

(1) performs only reads: $\mathtt{typ}(\mathtt{tlab}(\mu_\tau(\beta)(i))) = \mathsf{R}$ for $n \le i \le n'$; and
(2) returns the program to the starting state of the loop: $\mathtt{src}(\mu_\tau(\beta)(n)) = \mathtt{tgt}(\mu_\tau(\beta)(n'))$.

An *infinite spinloop* of thread τ in a behavior β is an infinite sequence s of consecutive spinloop iterations of thread τ (*i.e.*, $s(i) = [n_i, n'_i] \implies \exists n'_{i+1}.\, s(i+1) = [n'_i, n'_{i+1}]$).

If infinite spinloops are the only source of unbounded behavior in programs (*i.e.,* their individual iterations are of bounded length and there are boundedly many writes to each memory location), then because of fairness, an infinite spinloop has to eventually read from the mo-maximal writes.

Theorem 5.3. *Let β be a behavior of a deterministic program and G be a fair execution graph with $G.\mathsf{E} = \mathsf{Event}(\beta)$ and $G.\mathsf{sc}_{\mathsf{loc}}$ (see §3.2) irreflexive. For every infinite spinloop s of a thread τ in β whose iterations have bounded length and read only from locations that are written to by finitely many writes in G, there is a loop iteration $s(i)$ whose reads all read from mo-maximal writes.*

This theorem provides a sufficient condition for establishing termination of spinloops. In the supplementary material, we also establish the other direction: whenever a deterministic program has a behavior where all non-terminated threads end with a loop iteration reading from mo-maximal writes, then it has an infinite memory-fair behavior.

5.1 Spinlock

Consider the following spinlock implementation:

```
int l := 0
void lock() { int r
              repeat { repeat { r := l } until (r = 0) }
                until (CAS(l, 0, 1)) }
void unlock() { l := 0 }
```

Theorem 5.4. *All thread-fair behaviors of the following program under $\mathcal{G}^{\mathtt{fair}}_{\{\mathsf{SC},\mathsf{TSO},\mathsf{RA},\mathsf{StrongCOH}\}}$ are finite:*

$$\begin{matrix} lock() \\ unlock() \end{matrix} \Big\| \begin{matrix} lock() \\ unlock() \end{matrix} \Big\| \cdots \Big\| \begin{matrix} lock() \\ unlock() \end{matrix} \qquad \text{(SpinLock-Client)}$$

Proof. Assume for the sake of contradiction that the program has an infinite thread-fair behavior β, which is induced by a fair execution graph G. By inspection, since G is infinite, β must contain an infinite spinloop. The number of write events to the location l in G is finite since each thread makes at most two writes to l. Fix the mo-maximal one among them and denote it w. Due to thread fairness of β, the value written by w has to be 0. (Otherwise, it could have been only the value 1 produced by the **CAS** instruction, which is followed by a store writing 0, and the write event produced by the store would have been mo-following for w by {SC, TSO, RA, StrongCOH}-consistency of G.) By Theorem 5.3, there is a spinloop iteration that reads from w, which is a contradiction, since reading 0 from location l exits the loop. □

5.2 Ticket Lock

Consider the following ticket lock implementation:

```
int serving := 0, ticket := 0
void lock() { int s := 0, r := FADD(ticket, 1)
              repeat { s := serving } until (s = r) }
void unlock() { serving := serving + 1 }
```

Theorem 5.5. *In every thread-fair behavior of the following program under* $\mathcal{G}^{\mathtt{fair}}_{\{\mathsf{SC,TSO,RA,StrongCOH}\}}$, $r_1, \ldots, r_N$ *all grow unboundedly:*

$$\begin{array}{l|l|l|l} L_1 : lock() & L_2 : lock() & & L_N : lock() \\ r_1 := r_1 + 1 & r_2 := r_2 + 1 & \cdots & r_N := r_N + 1 \\ unlock() & unlock() & & unlock() \\ \textbf{goto}\ L_1 & \textbf{goto}\ L_2 & & \textbf{goto}\ L_N \end{array}$$

Proof. For any thread-fair behavior β of this program and a fair execution graph G inducing β, it can be shown that each call to *lock* reads a unique value from *ticket*, and that whenever a certain *lock* call reads ticket value v (and the spinloop exits), the corresponding *unlock* writes to *serving* value $v + 1$. Moreover, the values written to *ticket* and to *serving* are strictly increasing along $G.\mathtt{mo}$. (These are standard safety properties, so we elide details of their proofs.)

By means of contradiction, now assume that there is a fair execution graph G inducing β where r_i for some $1 \le i \le N$ is incremented only a finite number of times.

Due to thread-fairness of β, the only way this can happen is if thread i has an infinite spinloop. There may well be multiple threads with infinite spinloops, so among those threads let us consider the thread τ that reads the smallest value for *ticket*, say k, just before going into the infinite spinloop. So, for all $0 \le j < k$, some *lock* has incremented *ticket* to value j and subsequently *serving* to value $j + 1$. In particular, the $\mathtt{mo}$-maximal among those sets *serving* to value k. Note that there cannot be any writes to *serving* with larger values because they all require *serving* to first be set to $k + 1$ (which does not happen since τ is stuck in a spinloop).

Because of thread-fairness and Theorem 5.3, the infinite spinloop must have an iteration that reads from the $\mathtt{mo}$-maximal write to *serving*, *i.e.,* reading value k. This is a contradiction, because reading k exits the loop. □

5.3 MCS lock

As a third example, we study the MCS lock [Mellor-Crummey and Scott 1991], which is the basis of the `qspinlock` currently used in the Linux kernel and the highly scalable NUMA-aware HMCS lock [Chabbi et al. 2015]. For the latter, Oberhauser et al. [2021b] observe that "the fences necessary for the HMCS lock on systems with processors that use weak ordering" presented in the original HMCS paper [Chabbi et al. 2015, p. 218] result in non-terminating behaviors under RC11, which do in fact occur in practice when running the HMCS lock on a Kunpeng 920 Arm server. Non-termination is due to a missing release fence (or store-release) in the MCS lock used in that algorithm. For simplicity, we therefore limit our discussion to the MCS lock, whose code follows.

```
QNode Lock := null

void lock(QNode n) {
  n.locked := 1
  n.next := null
  // fence^rel missing in HMCS paper
  QNode pred := SWAP^acqrel(Lock, n)
  if pred ≠ null
  then pred.next := n
       while n.locked = 1 { }
       fence^acq
}
```

```
void unlock(QNode n) {
  fence^rel
  QNode succ := n.next
  fence^acq   // can be elided on ARM
  if succ = null
  then if CAS^acqrel(Lock, n, null)
       then return
       else repeat { succ := n.next }
            until succ ≠ null
  succ.locked := 0
}
```

The MCS lock uses a FIFO queue to ensure fairness. Therefore, the *lock* and *unlock* functions take a *QNode* argument to identify the calling thread. A thread T can enter the critical section (after

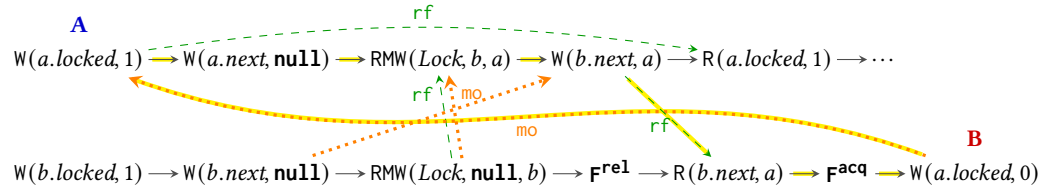
calling the *lock* function) either if the queue is empty or after its predecessor in the queue lowers the *locked* bit in T's *QNode*. To release the lock, a thread T lowers the *locked* bit of the next thread in the queue, or if no such thread exists, empties the queue.

Consider now the following client program, in which two threads enter the critical section once.

$$\begin{array}{l|l} a := \texttt{new QNode}() & b := \texttt{new QNode}() \\ lock(Lock, a) & lock(Lock, b) \\ unlock(Lock, a) & unlock(Lock, b) \end{array} \qquad \text{(MCS-Client)}$$

Suppose we want to show that this program terminates and, in particular, that the **while** loops in *lock* terminate if ever reached. Due to symmetry, we only consider the loop for $n = a$. By Theorem 5.3, it suffices to consider the iteration in which the loop reads from the mo-maximal store. We can now construct all candidate mos and attempt to show for each one that either the mo-maximal store allows the loop to terminate or any graph with that mo is not RC11-consistent.

It is easy to show that in every execution of this program in which that loop is reached, there are exactly two non-initial stores to $a.locked$, generated by the calls $lock(Lock, a)$ and $unlock(Lock, b)$, respectively. For brevity's sake, we call these stores **A** and **B** respectively. Since **B** writes $a.locked = 0$, reading from it allows the loop to terminate. Consequently, the loop may only diverge in execution graphs in which **A** is mo-maximal. Such a graph is shown below.



The graph is in fact RC11-consistent, and therefore the client program does not always terminate. Once, however, we add back the commented-out $\texttt{fence}^{\texttt{rel}}$ in the *lock* function, then the highlighted po ; rf ; po ; mo cycle in the execution graph above is forbidden. Similarly, the release fence also rules out all other graphs in which **A** is the mo-maximal store, and we can thus prove the following theorem. (Our Coq proof generalizes this theorem to an arbitrary finite number of threads.)

Theorem 5.6. *If the* $\texttt{fence}^{\texttt{rel}}$ *in the MCS lock is uncommented, MCS-Client's thread-fair behaviors under* $G^{\texttt{fair}}_{\{SC,TSO,RA,RC11\}}$ *are all finite.*

6 RELATED WORK AND DISCUSSION

We have investigated fairness in (po ∪ rf)-acyclic weak memory models, both operationally and declaratively, established four equivalence results, and showed how the declarative formulations can be used for reasoning about program termination.

Several papers, *e.g.,* [Bouajjani et al. 2014; Cerone et al. 2015; Gotsman and Burckhardt 2017], have studied declarative formulations of transactional consistency with prefix-finiteness constraints to ensure that a transaction is never preceded by an infinite set of other transactions. In particular, Gotsman and Burckhardt [2017] established a connection between declarative presentations that include fairness constraints and operational presentations for models in their "Global Operation Sequencing" framework. The TSO model can be expressed in this framework. Their declarative specifications require prefix-finiteness of the global visibility order, while we derive this property from prefix finiteness of more local relations (mo and fr). Thus, our formulation is easily applicable for model checking based on partial order reduction in the style of Kokologiannakis et al. [2017, 2019]. To the best of our knowledge, this is the first work to make a connection between liveness

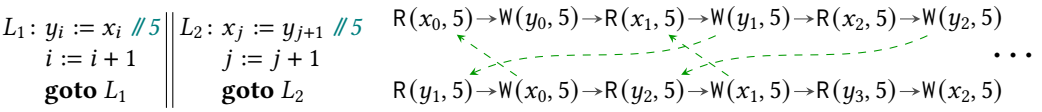
in declarative models formulated in the widely used framework of Alglave et al. [2014] and in operational models.

Termination of the MCS lock was previously studied by [Oberhauser et al. 2021a]; however, due to the lack of a formal definition of fairness, Oberhauser et al. [2021a] assumed a highly technical consequence of fairness in their proofs. Our unified definition of fairness and Theorem 5.3 bridge the gap left in their arguments and allow us to obtain the first complete formal termination proof for the MCS lock.

We note that our approach for establishing termination of spinloops is not only useful for manually proving deadlock-freedom and related progress properties as shown in §5, but can also be used to automatically establish termination of programs whose only potentially unbounded behavior is due to spinloops. One can use Theorem 5.3 to reason about the termination of such programs by examining only a finite number of finite execution graphs. This approach has actually been implemented in the GenMC model checker [Kokologiannakis and Vafeiadis 2021], and thus termination of the example programs in the paper (for a bounded number of threads) can also be shown automatically.

We outline two directions for future work, which concern extending our results to more complex models.

Fairness under non-*(po ∪ rf)*-acyclic models. Some low-level hardware memory models, such as Arm [Flur et al. 2016] and POWER [Alglave et al. 2014], and hardware-inspired memory models, such as LKMM [Alglave et al. 2018] and IMM [Podkopaev et al. 2019], record syntactic dependencies between instructions so as to allow certain executions with cycles in po ∪ rf. In these models, prefix-finiteness of mo and fr alone does not suffice for prefix-finiteness of the appropriate "happens-before" relation. For instance, under Arm (version 8) [Flur et al. 2016], assuming prefix-finiteness of mo and fr does not forbid the out-of-thin-air read of the value 5 in the following example (with an unbounded address domain):



We conjecture that the appropriate liveness condition for Arm is to require prefix-finiteness of the "ordered-before" (ob) relation. We leave adapting the operational Arm model to ensure fairness and establishing correspondence between the two models for future work.

Similarly, there are a number of more advanced memory models for programming languages that aim to admit write-after-read reorderings (and thus have to allow (po ∪ rf) cycles) such as JMM [Manson et al. 2005], Promising [Kang et al. 2017], Pomsets with Preconditions [Jagadeesan et al. 2020], and Weakestmo [Chakraborty and Vafeiadis 2019]. Integrating liveness requirements in such memory models is left for future work.

Weak RMWs. Besides ordinary ("strong") CAS instructions, C11 supports "weak" CASes,[7] which may fail spuriously, *i.e.,* even when they read the expected value, since on some architectures—namely, POWER and Arm—weak CASes are more efficient than strong ones. A strong CAS can be implemented by repeatedly performing a weak CAS in a loop as long as it fails spuriously. Termination of such loops depends upon the weak CASes not always failing spuriously, which constitutes an additional fairness requirement. Since this requirement is orthogonal to the notion of memory fairness introduced in this paper, we leave it for future work.

[7] See https://en.cppreference.com/w/cpp/atomic/atomic/compare_exchange [accessed November-2020].

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their helpful feedback. This research was supported in part by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement no. 851811 and 101003349). Lahav was also supported by the Israel Science Foundation (grant number 1566/18) and by the Alon Young Faculty Fellowship.

REFERENCES

Jade Alglave, Luc Maranget, Paul E. McKenney, Andrea Parri, and Alan S. Stern. 2018. Frightening Small Children and Disconcerting Grown-ups: Concurrency in the Linux Kernel. In *ASPLOS 2018*. ACM, 405–418. https://doi.org/10.1145/3173162.3177156

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

Mark Batty, Kayvan Memarian, Scott Owens, Susmit Sarkar, and Peter Sewell. 2012. Clarifying and Compiling C/C++ Concurrency: From C++11 to POWER. In *POPL*. ACM, New York, NY, USA, 509–520. https://doi.org/10.1145/2103656.2103717

Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ Concurrency. In *POPL*. ACM, New York, NY, USA, 55–66. https://doi.org/10.1145/1926385.1926394

John Bender and Jens Palsberg. 2019. A Formalization of Java's Concurrent Access Modes. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 142 (Oct. 2019), 28 pages. https://doi.org/10.1145/3360568

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, and Jad Hamza. 2014. Verifying Eventual Consistency of Optimistic Replication Systems. In *POPL*. ACM, New York, NY, USA, 285–296. https://doi.org/10.1145/2535838.2535877

Ahmed Bouajjani, Constantin Enea, Suha Orhun Mutluergil, and Serdar Tasiran. 2018. Reasoning About TSO Programs Using Reduction and Abstraction. In *CAV*. Springer International Publishing, Cham, 336–353. https://doi.org/10.1007/978-3-319-96142-2_21

Ahmed Bouajjani, Roland Meyer, and Eike Möhlmann. 2011. Deciding Robustness against Total Store Ordering. In *ICALP*. Springer Berlin Heidelberg, Berlin, Heidelberg, 428–440. https://doi.org/10.1007/978-3-642-22012-8_34

Andrea Cerone, Giovanni Bernardi, and Alexey Gotsman. 2015. A Framework for Transactional Consistency Models with Atomic Visibility. In *CONCUR*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 58–71. https://doi.org/10.4230/LIPIcs.CONCUR.2015.58

Milind Chabbi, Michael Fagan, and John Mellor-Crummey. 2015. High Performance Locks for Multi-Level NUMA Systems. In *PPoPP*. ACM, New York, NY, USA, 215–226. https://doi.org/10.1145/2688500.2688503

Soham Chakraborty and Viktor Vafeiadis. 2019. Grounding thin-air reads with event structures. *Proc. ACM Program. Lang.* 3, POPL (2019), 70:1–70:28. https://doi.org/10.1145/3290383

Egor Derevenetc and Roland Meyer. 2014. Robustness against Power is PSpace-complete. In *ICALP*. Springer, Berlin, Heidelberg, 158–170. https://doi.org/10.1007/978-3-662-43951-7_14

Stephen Dolan, KC Sivaramakrishnan, and Anil Madhavapeddy. 2018. Bounding Data Races in Space and Time. In *PLDI*. ACM, New York, NY, USA, 242–255. https://doi.org/10.1145/3192366.3192421

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, NY, USA, 608–621. https://doi.org/10.1145/2837614.2837615

Nissim Francez. 1986. *Fairness*. Springer. https://doi.org/10.1007/978-1-4612-4886-6

Alexey Gotsman and Sebastian Burckhardt. 2017. Consistency Models with Global Operation Sequencing and their Composition. In *DISC*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 23:1–23:16. https://doi.org/10.4230/LIPIcs.DISC.2017.23

Radha Jagadeesan, Alan Jeffrey, and James Riely. 2020. Pomsets with preconditions: A simple model of relaxed memory. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 194:1–194:30. https://doi.org/10.1145/3428262

Jan-Oliver Kaiser, Hoang-Hai Dang, Derek Dreyer, Ori Lahav, and Viktor Vafeiadis. 2017. Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris. In *ECOOP*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 17:1–17:29. https://doi.org/10.4230/LIPIcs.ECOOP.2017.17

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

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
Michalis Kokologiannakis, Azalea Raad, and Viktor Vafeiadis. 2019. Model Checking for Weakly Consistent Libraries. In *PLDI 2019*. ACM, New York, NY, USA, 96–110. https://doi.org/10.1145/3314221.3314609
Michalis Kokologiannakis and Viktor Vafeiadis. 2021. GenMC: A Model Checker for Weak Memory Models. In *CAV 2021 (LNCS, Vol. 12759)*. Springer, 427–440. https://doi.org/10.1007/978-3-030-81685-8_20
Ori Lahav, Nick Giannarakis, and Viktor Vafeiadis. 2016. Taming Release-Acquire Consistency. In *POPL*. ACM, New York, NY, USA, 649–662. https://doi.org/10.1145/2837614.2837643
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, Egor Namakonov, Jonas Oberhauser, Anton Podkopaev, and Viktor Vafeiadis. 2021a. Making Weak Memory Models Fair. Full paper version with appendices. arXiv:2012.01067 [cs.PL]
Ori Lahav, Egor Namakonov, Jonas Oberhauser, Anton Podkopaev, and Viktor Vafeiadis. 2021b. *Making Weak Memory Models Fair: OOPSLA 2021 artifact*. https://doi.org/10.5281/zenodo.5496483
Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. 2017. Repairing Sequential Consistency in C/C++11. In *PLDI*. ACM, New York, NY, USA, 618–632. https://doi.org/10.1145/3062341.3062352
Leslie Lamport. 1977. Proving the Correctness of Multiprocess Programs. *IEEE Trans. Software Eng.* 3, 2 (1977), 125–143. https://doi.org/10.1109/TSE.1977.229904
Leslie Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Computers* 28, 9 (1979), 690–691. https://doi.org/10.1109/TC.1979.1675439
D. Lehmann, A. Pnueli, and J. Stavi. 1981. Impartiality, Justice and Fairness: The Ethics of Concurrent Termination. In *ICALP*. Springer Berlin Heidelberg, Berlin, Heidelberg, 264–277. https://doi.org/10.1007/3-540-10843-2_22
Jeremy Manson, William Pugh, and Sarita V. Adve. 2005. The Java Memory Model. In *POPL 2005*. ACM, New York, 378–391. https://doi.org/10.1145/1040305.1040336
Roy Margalit and Ori Lahav. 2021. Verifying Observational Robustness against a C11-Style Memory Model. *Proc. ACM Program. Lang.* 5, POPL, Article 4 (Jan. 2021), 33 pages. https://doi.org/10.1145/3434285
John M. Mellor-Crummey and Michael L. Scott. 1991. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Trans. Comput. Syst.* 9, 1 (Feb. 1991), 21–65. https://doi.org/10.1145/103727.103729
Jonas Oberhauser. 2018. Store Buffer Reduction in the Presence of Mixed-Size Accesses and Misalignment. In *VSTTE 2018 (LNCS, Vol. 11294)*. Springer, 322–344. https://doi.org/10.1007/978-3-030-03592-1_19
Jonas Oberhauser, Rafael Lourenco de Lima Chehab, Diogo Behrens, Ming Fu, Antonio Paolillo, Lilith Oberhauser, Koustubha Bhat, Yuzhong Wen, Haibo Chen, Jaeho Kim, and Viktor Vafeiadis. 2021a. VSync: Push-Button Verification and Optimization for Synchronization Primitives on Weak Memory Models. In *ASPLOS*. ACM, New York, NY, USA, 530–545. https://doi.org/10.1145/3445814.3446748
Jonas Oberhauser, Lilith Oberhauser, Antonio Paolillo, Diogo Behrens, Ming Fu, and Viktor Vafeiadis. 2021b. Verifying and Optimizing the HMCS Lock for Arm Servers. In *NETYS 2021*. 16 pages. https://people.mpi-sws.org/~viktor/papers/netys2021-hmcs.pdf
Scott Owens, Susmit Sarkar, and Peter Sewell. 2009. A Better x86 Memory Model: x86-TSO. In *TPHOLs 2009 (LNCS, Vol. 5674)*. Springer, 391–407. https://doi.org/10.1007/978-3-642-03359-9_27
David Michael Ritchie Park. 1979. On the Semantics of Fair Parallelism. In *Abstract Software Specifications 1979 (LNCS, Vol. 86)*, Dines Bjørner (Ed.). Springer, 504–526. https://doi.org/10.1007/3-540-10007-5_47
Anton Podkopaev, Ori Lahav, and Viktor Vafeiadis. 2019. Bridging the Gap between Programming Languages and Hardware Weak Memory Models. *Proc. ACM Program. Lang.* 3, POPL, Article 69 (Jan. 2019), 31 pages. https://doi.org/10.1145/3290382
Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. 2017. Simplifying ARM Concurrency: Multicopy-Atomic Axiomatic and Operational Models for ARMv8. *Proc. ACM Program. Lang.* 2, POPL, Article 19 (Dec. 2017), 29 pages. https://doi.org/10.1145/3158107
Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. 2010. x86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors. *Commun. ACM* 53, 7 (2010), 89–97. https://doi.org/10.1145/1785414.1785443
Viktor Vafeiadis, Thibaut Balabonski, Soham Chakraborty, Robin Morisset, and Francesco Zappa Nardelli. 2015. Common Compiler Optimisations Are Invalid in the C11 Memory Model and What We Can Do about It. In *POPL*. ACM, New York, NY, USA, 209–220. https://doi.org/10.1145/2676726.2676995
Conrad Watt, Christopher Pulte, Anton Podkopaev, Guillaume Barbier, Stephen Dolan, Shaked Flur, Jean Pichon-Pharabod, and Shu-yu Guo. 2020. Repairing and Mechanising the JavaScript Relaxed Memory Model. In *PLDI*. ACM, New York, NY, USA, 346–361. https://doi.org/10.1145/3385412.3385973
Conrad Watt, Andreas Rossberg, and Jean Pichon-Pharabod. 2019. Weakening WebAssembly. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 133 (Oct. 2019), 28 pages. https://doi.org/10.1145/3360559