Overcoming Memory Weakness with Unified Fairness

Systematic verification of liveness in Weak Memory Models

Parosh Aziz Abdulla[1], Mohamed Faouzi Atig[1], Adwait Godbole[2],
Shankaranarayanan Krishna[3], and Mihir Vahanwala[4]

[1] Uppsala University `{parosh,mohamed_faouzi.atig}@it.uu.se`
[2] University of California, Berkeley `adwait@berkeley.edu`,
[3] IIT Bombay `krishnas@cse.iitb.ac.in`
[4] MPI-SWS, Saarbrücken `mvahanwa@mpi-sws.org`

Abstract. We consider the verification of liveness properties for concurrent programs running on weak memory models. To that end, we identify notions of fairness that preclude demonic non-determinism, are motivated by practical observations, and are amenable to algorithmic techniques. We provide both logical and stochastic definitions of our fairness notions, and prove that they are equivalent in the context of liveness verification. In particular, we show that our fairness allows us to reduce the liveness problem (repeated control state reachability) to the problem of simple control state reachability. We show that this is a general phenomenon by developing a uniform framework which serves as the formal foundation of our fairness definition, and can be instantiated to a wide landscape of memory models. These models include SC, TSO, PSO, (Strong/Weak) Release-Acquire, Strong Coherence, FIFO-consistency, and RMO.

1 Introduction

Safety and liveness properties are the cornerstones of concurrent program verification. While safety and liveness are complementary, verification methodologies for the latter tend to be more complicated for two reasons. First, checking safety properties, in many cases, can be reduced to the (simple) reachability problem, while checking liveness properties usually amounts to checking repeated reachability of states [49]. Second, concurrency comes with an inherent *scheduling non-determinism*, i.e., at each step, the scheduler may non-deterministically select the next process to run. Therefore, liveness properties need to be accompanied by appropriate fairness conditions on the scheduling policies to prohibit trivial blocking behaviors [42]. In the example of two processes trying to acquire a lock, demonic non-determinism [20] may always favour one process over the other, leading to starvation.

Despite the gap in complexity, the verification of liveness properties has attracted much research in the context of programs running under the classical Sequential Consistency (SC) [40]. An execution of a program under SC is a non-deterministically chosen interleaving of its processes' atomic operations. A write by any given process is immediately visible to all other processes, and reads are made from the most recent write to the memory location in question. SC is (relatively) simple since the only non-determinism comes from interleaving.

Weak memory models forego the fundamental SC guarantee of immediate visibility of writes to optimize for performance. More precisely, a write operation by a process may asynchronously be propagated to the other processes. The delay could be owed to physical buffers or caches, or could simply be a virtual one thanks to instruction reorderings allowed by the semantics of the programming language. Hence we have to contend with a (potentially unbounded) number of write operations that are "in transit", i.e., they have been issued by a process but they have yet to reach the other processes. In this manner, weak memory introduces a second source of non-determinism, namely *memory non-determinism*, reflecting the fact that write operations are non-deterministically (asynchronously) propagated to the different processes. Formal models for weak memory, ranging from declarative models [7,41,39,21,35] to operational ones [44,48,15,30] make copious use of non-determinism (non-determinism over entire executions in the case of declarative models and non-deterministic transitions in the case of operational models). While we have seen extensive work on verifying safety properties for program running under weak memory models, the literature on liveness for programs running under weak memory models is relatively sparse, and it is only recently we have seen efforts in that direction [5,36].

As mentioned earlier, we need fairness conditions to exclude demonic behaviors when verifying liveness properties. A critical issue here is to come up with an *appropriate* fairness condition, i.e., a condition that (i) is sufficiently strong to eliminate demonic non-determinism and (ii) is sufficiently weak to allow all "good" program behaviors. To illustrate the idea, let us go back to the case of SC. Here, traditional fairness conditions on processes, such as *strong fairness* [31], are too weak if interpreted naively, e.g. "along any program run, each process is scheduled infinitely often". The problem is that even though a strongly fair scheduler may pick a process infinitely often, it may choose to do so only in configurations where the process cannot progress since its guards are not satisfied. Such guards may, for instance, be conditions on the values of the shared variables. For example, executions of the program in Fig. 1 may not terminate under SC, since the second process may only get scheduled when the value of x is 2, thereby looping infinitely around the `do-while` loop.

Stronger fairness conditions such as transition fairness, and probabilistic fairness [10,27] can help avoid this problem. They imply that any *transition* enabled infinitely often is also taken infinitely often (with probability one in the case of probabilistic fairness). Transition fairness eliminates demonic scheduler non-determinism, and hence it is an appropriate notion of fairness in the case of

```
r = 0;                              ‖  do { s = x; } until (s == 1)
while (r != 1)  {                   ‖  y = 1;
    x = 1; x = 2; r = y;            ‖
}                                   ‖
```

Fig. 1: Does this program always terminate? Only if we can guarantee that the process to the right will eventually be scheduled to read when x = 1.

SC. However, it is unable to eliminate demonic memory non-determinism. The reason is that transition fairness allows runs of the programs where write events occur at a higher frequency than the frequency in which they are propagated to the processes. This means that, in the long run, a process may only see its own writes, potentially preventing its progress and, therefore, the system's progress as a whole. This scenario is illustrated in Figure 2.

```
do { x = 1; }                       ‖  do { x = 2; }
    until (x = 2 or y = 1);         ‖      until (x = 1 or y = 1);
y = 1;                              ‖  y = 1;
```

Fig. 2: This program is guaranteed to terminate under any model only if pending propagation is guaranteed to not accumulate unboundedly: e.g. in TSO, each process may never see the other's writes due to an overflowing buffer.

To deal with memory non-determinism, we exploit the fact that the sizes of physical buffers or caches are bounded, and instruction reorderings are bounded in scope. Therefore, in any practical setting, the number of writes in transit at a given moment cannot be unbounded indefinitely. This is what we seek to capture in our formalism. Based on this observation, we propose three new notions of fairness that (surprisingly) all turn out to be equivalent in the context of liveness. First, we introduce *boundedness fairness* which only considers runs of the system for which there is a bound b on the number of events in transit, in each configuration of the run. Note that the value of b is arbitrary (but fixed for a given run). Bounded fairness is apposite: (i) it is sufficiently strong to eliminate demonic memory non-determinism, and (ii) it is sufficiently weak to allow all reasonable behaviors (as mentioned above, practical systems will bound the number of transient messages). Since we do not fix the value of the bound, this allows parameterized reasoning, e.g., about buffers of any size: our framework does not depend on the actual value of the bound, only on its mere existence. Furthermore, we define two additional related notions of fairness for memory non-determinism. The two new notions rely on *plain configurations*: configurations in which there are no transient operations (all the writes operations have reached all the processes). First, we consider *plain fairness*: along each infinite run, the set of plain configurations is visited infinitely often, and then define the

probabilistic version: each run almost surely visits the set of plain configurations. We show that the three notions of fairness are equivalent (in §4, we make precise the notion of equivalence we use).

After we have defined our fairness conditions, we turn our attention to the verification problem. We show that verifying the repeated reachability under the three fairness conditions, for a given memory model m, is reducible to the simple reachability under m. Since our framework does not perform program transformations we can prove liveness properties for program P through proving simple reachability on the same program P. As a result we obtain two important sets of corollaries: if the simple reachability problem is decidable for m, then the repeated reachability problem under the three fairness conditions are also decidable. This is the case when the memory model m is TSO, PSO, SRA, etc. Even when the simple reachability problem is not decidable for m, e.g., when m is RA, RMO, we have still succeeded to reduce the verification of liveness properties under fairness conditions to the the verification of simple probability. This allows leveraging proof methodologies developed for the verification of safety properties under these weak memory models (e.g., [29,22]).

Having identified the fairness conditions and the verification problem, there are two potential approaches, each with its advantages and disadvantages. We either instantiate a framework for individual memory models one after one or define a general framework in which we can specify multiple memory models and apply the framework "once for all". The first approach has the benefit of making each instantiation more straightforward, however, we always need to translate our notion of fairness into the specific formulation. In the second approach, although we incur the cost of heavier machinery, we can subsequently take for granted the fact that the notion of fairness is uniform across all models, and coincides with our intuition. This allows us to be more systematic in our quest to verify liveness. In this paper, we have thus chosen to adopt the second approach. We define a general model of weak memory models in which we represent write events as sequences of messages ordered per variable and process. We augment the message set with additional conditions describing which messages have reached which processes. We use this data structure to specify our fairness conditions and solve our verification problems. We instantiate our framework to apply our results to a wide variety of memory models, such as RMO [12], FIFO consistency, RA, SRA, WRA [35,34], TSO [13], PSO, StrongCOH (the relaxed fragment of RC11) [30], and SC [40].

In summary, we make the following contributions

- Define new fairness conditions that eliminate demonic memory non-determinism.

- Reduce checking the repeated reachability problem under these fairness conditions to the simple reachability problem.

- Introduce a general formalism for weak memory models that allows applying our results uniformly to a broad class of memory models.

– Prove the decidability of liveness properties for models such as TSO, PSO, SRA, WRA, StrongCOH, and opening the door for leveraging existing proof frameworks for simple reachability for other models such as RA.

We give an overview of a wide landscape of memory models in §3.3, and provide a high level explanation of the versatility of our framework. We refer the reader to Appendix A for detailed instantiations.

Structure of the paper. We begin by casting concurrent programs as transition systems in §2. In §3, we develop our framework for the memory such that the desired fairness properties can be meaningfully defined across several models. In §4, we define useful fairness notions and prove their equivalence. Finally, in §5 we show how the liveness problems of repeated control state reachability reduce to the safety problem of control state reachability, and obtain decidability results.

2 Modelling Concurrent Programs

We consider concurrent programs as systems where a set of processes run in parallel, computing on a set of process-local variables termed as *registers* and communicating through a set of *shared variables*. This inter-process communication, which consists of reads from, writes to, and atomic compare-and-swap operations on shared variables, is mediated by the *memory subsystem*. The overall system can be visualized as a composition of the process and memory subsystems working in tandem. In this section we explain how concurrent programs naturally induce *labelled transition systems*.

2.1 Labelled Transition Systems

A labelled transition system is a tuple $\mathcal{T} = \langle \Gamma, \rightarrow, \Lambda \rangle$ where Γ is a (possibly-infinite) set of configurations, $\rightarrow \subseteq \Gamma \times \Lambda \times \Gamma$ is a transition relation, and Λ is the set of labels that annotate transitions. We also refer to them as annotations to disambiguate from instruction labels. We write $\gamma \xrightarrow{l} \gamma'$ to denote that $(\gamma, l, \gamma') \in \rightarrow$, in words that there is a transition from γ to γ' with label l. We denote the transitive closure of $\rightarrow$ by $\xrightarrow{*}$, and the k-fold self-composition (for $k \in \mathbb{N}$) as $\xrightarrow{k}$.

Runs and paths A (possibly infinite) sequence of valid transitions $\rho = \gamma_1 \rightarrow \gamma_2 \rightarrow \gamma_3 \cdots$ is called a run. We say that a run is a γ-run if the initial configuration in the run is γ, and denote the set of γ-runs as $\mathsf{Runs}(\gamma)$. We call a (finite) prefix of a run as a *path*. In some cases transition systems are *initialized*, i.e. an initial set $\Gamma_{\mathsf{init}} \subseteq \Gamma$ is specified. In such cases, we call runs starting from some initial configuration ($\gamma_1 \rightarrow \gamma_2 \rightarrow \gamma_3 \ldots$ with $\gamma_1 \in \Gamma_{\mathsf{init}}$) as initialized runs.

2.2 Concurrent Programs

The sequence of instructions executed by each process is dictated by a concurrent program, which induces a *process subsystem*. We begin by formulating the notion of a program. We assume a finite set $\mathbb{P}$ of processes that operate over a (finite) set $\mathbb{X}$ of shared variables. Fig. 3 gives the grammar for a small but general assembly-like language that we use for defining the syntax of concurrent programs. A program instance, prog is described by a set of shared variables, $\mathtt{var}^*$, followed by the codes of the processes, $(\mathtt{proc}\ \mathtt{reg}^*\ \mathtt{instr}^*)^*$. Each process $p \in \mathbb{P}$ has a finite set $\mathsf{Regs}(p)$ of (local) *registers*. We assume w.l.o.g. that the sets of registers of the different processes are disjoint, and define $\mathsf{Regs}(\mathsf{prog}) := \cup_{p \in \mathbb{P}} \mathsf{Regs}(p)$. We assume that the data domain of both the shared variables and registers is a finite set $\mathbb{D}$, with a special element $0 \in \mathbb{D}$. The code of a process starts by declaring its set of registers, $\mathtt{reg}^*$, followed by a sequence of instructions.

```
prog  ::= var* (proc reg* instr*)*
instr ::= lbl : stmt
stmt  ::= var:=reg | reg:=var | reg:=CAS(var,reg,reg) |
reg:=expr | if reg then lbl | term
```

Fig. 3: A simple programming language.

An instruction i is of the form l : stmt where l is a unique (across all processes) instruction label that identifies the instruction, and stmt is a statement. The labels comprise the set of values the program counters of the processes may take. The problems of (repeated) instruction label reachability, which ask whether a section of code is accessed (infinitely often), are of importance to us.

Read (reg := var) and write (var : = reg) statements read the value of a shared variable into a register, and write the value of a register to a shared variable respectively. The CAS statement is the *compare-and-swap* operation which atomically executes a read followed by a write. We consider a non-blocking version of the CAS operation which returns a boolean indicating whether the operation was successful (the expected value was read and atomically updated to the new value). The write is performed only if the read matches the expected value.

We assume a set expr of expressions containing a set of operators applied to constants and registers without referring to the shared variables. The reg := expr statement updates the value of register reg by evaluating expression expr. We exact set of expressions is orthogonal to our treatment, and hence left uninterpreted. The if-statement has its usual interpretation, and control flow commands such as while, for, and goto-statements, can be encoded with branching and if-statements as usual.

2.3 Concurrent Programs as Labelled Transition Systems

We briefly explain the abstraction of a concurrent program as a labelled transition system. The details for the process component, i.e. evolution of the program counter and register contents, follow naturally. The key utility of this approach lies in the modelling of the memory subsystem, which we devote §3 to.

Configurations A configuration γ is expressed as a tuple $\langle (L, R), \gamma_{\mathtt{m}} \rangle$, where L maps processes to their current program counter values, R maps registers to their current values, and $\gamma_{\mathtt{m}}$ captures the current state of the (weak) memory.

Transitions In our model, a step in our system is either: (a) a silent memory update, or (b) a process executing its current instruction. In case (a), only the memory component $\gamma_{\mathtt{m}}$ of γ changes. The relation is governed by the definition of the memory subsystem. In case (b), if the instruction is the terminal one, or assigns an expression to a register, or a conditional, then only the process component (L, R) of γ changes. Here, the relation is obvious. Otherwise, the two components interact via a read, write or CAS, and both undergo changes. Here again, the relation is governed by what the memory subsystem permits.

Annotations Silent memory update steps are annotated with $\mathtt{m} : \mathtt{Upd}$. Transitions involving process p executing an instruction that does not involve memory are annotated with $p : \bot$. On the other hand, $p : \mathtt{R}(\mathsf{x}, \mathsf{d})$, $p : \mathtt{W}(\mathsf{x}, \mathsf{d})$, $p : \mathtt{CAS}(\mathsf{x}, \mathsf{d}, \mathsf{d}', \mathsf{b})$ represent reads, writes and CAS operations by p respectively. The annotations indicate the variable and the associated values.

To study this transition system, one must understand which transitions, annotated thus, are enabled. For this, it is clear that we must delve into the details of the memory subsystem.

3 A Unified Framework for Weak Memory Models

In this section, we present our unified framework for representing weak memory models. We begin by describing the modelling aspects of our framework at a high level.

We use a message-based framework, where each write event generates a *message*. A process can use a write event to justify its read only if the corresponding message has been *propagated* to it by the memory subsystem. The total chronological order in which a process p writes to variable x is given by poloc (per-location program order). We work with models where the order of propagation is consistent with poloc. This holds for several models of varying strengths. This requirement allows us to organise messages into per-variable, per-process *channels*. We discuss these aspects of the framework in §3.1. Weak memory models define additional causal dependencies over poloc. Reading a message may cause other messages it is dependent on to become illegible. We discuss our mechanism to capture

these dependencies in §3.2. The strength of the constraints levied by causal dependencies varies according to memory model. In §3.3, we briefly explain how our framework allows us to express causality constraints of varying strength, by considering a wide landscape of weak memory models. We refer the reader to Appendix A for the technical details of the instantiations.

3.1 Message structures

Message. A write by a process to a variable leads to the formation of a *message*, which, first and foremost records the value being written. In order to ensure atomicity, a message also records a boolean denoting whether the message can be used to justify the read of an *atomic* read-write operation, i.e. CAS. Finally, to help with the tracking of causal dependencies generated by read events, a message records a set of processes seen $\subseteq \mathbb{P}$ that have sourced a read from it. Thus, a message is a triple and we define the set of messages as: $\mathsf{Msgs} = \mathbb{D} \times \mathbb{B} \times 2^{\mathbb{P}}$.

Channels. A *channel* $\mathsf{e}(\mathsf{x}, p)$ is the sequence of messages corresponding to writes to x by process p. The total poloc order of these writes naturally induces the channel order. By design, we will ensure that the configuration holds finitely many messages in each channels. We model each channel as a word over the message set: $\mathsf{e}(\mathsf{x}, p) \in \mathsf{Msgs}^*$. A message structure is a collection of these channels: $\mathsf{e} : \mathbb{X} \times \mathbb{P} \to \mathsf{Msgs}^*$.

3.2 Ensuring consistency of executions

Memory models impose constraints restricting the set of message that can be read by a process. The framework uses state elements frontier, source, constraint that help enforce these constraints. These elements reference positions within each channel which is something that we now discuss.

Channel positions. The channel order provides the order of propagation of write messages to any process (which in turn is aligned with poloc). Thus, for any process p', channel $\mathsf{e}(\mathsf{x}, p)$ is partitioned into a prefix of messages that are outdated, a null or singleton set of messages that can be used to justify a read, and a suffix of messages that are yet to be propagated. In order to express these partitions, we need to identify not only nodal positions, but also to *internodes* (space between nodes). To this end, we index channels using the set $\mathbb{W} = \mathbb{N} \cup \mathbb{N}^+$. Positions indexed by $\mathbb{N}$ denote nodal positions (with a message), while positions indexed with $\mathbb{N}^+$ denote internodes. For a channel of length n, the positions are ordered as: $\top = 0^+ < 1 < 1^+ < 2 \cdots < n < n^+ = \bot$. A process can read from the message located at $\mathsf{e}(\cdot, \cdot)[i]$ for $i \in \mathbb{N}$.

Frontier. With respect to a given process, a message can either have been propagated but not readable, propagated and readable, or none. Since the propagation order of messages follows channel order, the propagated set
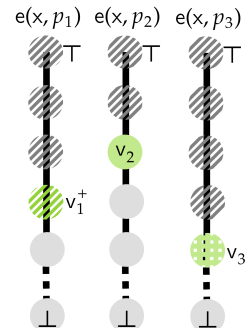

Fig. 4: Frontier and

of messages forms a prefix of the channel. This prefix-partitioning is achieved by a map frontier : $\mathbb{P} \times \mathbb{X} \times \mathbb{P} \to \mathbb{W}$. If frontier$(p, \cdot, \cdot)$ is an internode (of form i^+) then the message $\mathsf{v} = \mathsf{e}[i]$ has been propagated to p, but cannot be read because it is outdated. On the other hand, if frontier$(p, \cdot, \cdot) = i \in \mathbb{N}$, then the message $\mathsf{e}[i]$ can be read by the process. In Fig. 4, frontier$(p_1, \mathsf{x}, p_1/p_2/p_3)$ equal $\mathsf{v}_1^+/\mathsf{v}_2/\mathsf{v}_3$ respectively (the colored nodes). Consequently, the message at index v (or the ones before it) are unreadable (denoted by the pattern). On the other hand the messages at $\mathsf{v}_2, \mathsf{v}_3$ are readable.

Source. Given process p and variable x, the process potentially can source the read from any of the $|\mathbb{P}|$ channels on x. The second state element, source : $\mathbb{P} \times \mathbb{X} \to \mathbb{P}$ performs arbitration over this choice of read sources: p can read v only if $\mathsf{v} = \mathsf{frontier}(p, \mathsf{x}, \mathsf{source}(p, \mathsf{x}))$. In Fig. 4, while both nodes $\mathsf{v}_2, \mathsf{v}_3$ are not outdated, source$(p_1, \mathsf{x}) = p_3$, making v_3 the (checkered) node which p_1 reads from.

Constraint. The constraint element tracks causal dependencies between messages. For each message m, and channel, it identifies the last message on the channel that is a causal predecessor of m. It is defined as a map constraint : $\mathbb{N} \times \mathbb{X} \times \mathbb{P} \to \mathbb{W}$. Fig. 5 illustrates possible constraint$(\mathsf{v}_3, \cdot, \cdot)$ pointers for message node v_3 in the context of the channel configuration in Fig. 4.
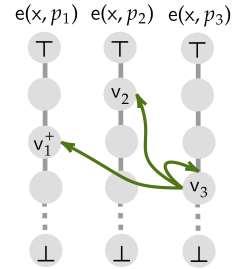
Fig. 5: Constraint.

Garbage collection. The frontier state marks the last messages in each channel that can be read by a process. Messages that are earlier than the frontier of all processes can be effectively eliminated from the system since they are illegible. We call this garbage collection (denoted as GC).

The overall memory configuration,

$$\gamma_{\mathbb{m}} = \langle \mathsf{e}, \underbrace{(\mathbb{P} \times \mathbb{X} \times \mathbb{P} \to \mathbb{W})}_{\text{frontier}}, \underbrace{(\mathbb{P} \times \mathbb{X} \to \mathbb{P})}_{\text{source}}, \underbrace{(\mathbb{V} \times \mathbb{X} \times \mathbb{P} \to \mathbb{W})}_{\text{constraint}} \rangle$$

consists of the message structure along with the consistency enforcing state.

Read transition. Our framework allows a unified read transition relation which is independent of the memory model that we work with. We now discuss this transition rule which is given in Fig. 6. Suppose process p is reading from variable x. First, we identify the arbitrated process p_s which is read from using the source state. Then we pick the message on the (x, p_s) channel which the frontier of p points to. Note that this must be a node of form $\mathbb{N}$. The read value is the value in this message. Finally, we update the frontier$(p, \cdot, \cdot)$ to reflect the fact that all messages in the causal prefix of the read message have propagated to p.

$$p_s = \gamma.\mathsf{source}(p, \mathsf{x}) \qquad \mathsf{v} = \gamma.\mathsf{frontier}(p, \mathsf{x}, p_s) \qquad \mathsf{v}.\mathsf{value} = \mathsf{d}$$
$$\gamma_1 = \gamma_1[\mathsf{v}.\mathsf{seen} \leftarrow \mathsf{v}.\mathsf{seen} \cup \{p\}]$$
$$\gamma_2 = \mathsf{GC}(\gamma_1[\lambda \mathsf{y}.\lambda p'.\ \mathsf{frontier}(p, \mathsf{y}, p') \leftarrow \max(\mathsf{frontier}(p, \mathsf{y}, p'), \mathsf{constraint}(\mathsf{v}, \mathsf{y}, p'))])$$
$$\gamma \xrightarrow{\ p:\mathsf{R}(\mathsf{x},\mathsf{d})\ }_\mathtt{m} \gamma_2$$

Fig. 6: The **read** transition, common to all models across the framework. For $\gamma \in \Gamma_\mathtt{m}$, $\gamma.\mathsf{frontier}, \gamma.\mathsf{source}, \gamma.\mathsf{constraint}$ represent the respective components of γ. For a node $\mathsf{v} \in \mathsf{Msgs}$, $\mathsf{v}.\mathsf{value} \in \mathbb{D}$ represents the written value in the message node v.

Example 1 (Store Buffer (SB)). Figure 7 shows the Store Buffer (SB) litmus test. The annotated outcome of store buffering is possible in all WRA/RA/S-RA/TSO models. Right after p_1 (resp. p_2) has performed both its writes to x (resp. y), we have $\mathsf{e}(\mathsf{y}, p_2) = \top \mathsf{v}_\mathsf{y}^0 \mathsf{v}_\mathsf{y}^1 \bot$, and $\mathsf{e}(\mathsf{x}, p_1) = \top \mathsf{v}_\mathsf{x}^0 \mathsf{v}_\mathsf{x}^1 \bot$.

This example illustrates how weak memory models allow non-deterministic delays the propagation of messages. In this example, $\mathsf{frontier}(p_2, \mathsf{x}, p_1) = \mathsf{v}_\mathsf{x}^0$, and $\mathsf{frontier}(p_1, \mathsf{y}, p_2) = \mathsf{v}_\mathsf{y}^0$, both processes see non-recent messages. On the other hand, the annotated outcomes are observed if $\mathsf{source}(p_1, \mathsf{y}) = p_2$ and $\mathsf{source}(p_2, \mathsf{x}) = p_1$.

$$
\begin{array}{c||c}
p_1 & p_2 \\
\mathtt{x = 0}\ \ //\ \mathsf{v}_\mathsf{x}^0 & \mathtt{y = 0}\ \ //\ \mathsf{v}_\mathsf{y}^0 \\
\mathtt{x = 1}\ \ //\ \mathsf{v}_\mathsf{x}^1 & \mathtt{y = 1}\ \ //\ \mathsf{v}_\mathsf{y}^1 \\
\mathtt{r = y}\ \ //\ 0 & \mathtt{s = x}\ \ //\ 0
\end{array}
$$

Fig. 7: SB

We now turn to a toy example (Figure 8) to illustrate the dependency enforcing and book-keeping mechanisms we have introduced.

Example 2. Consider an program with two shared variables, x, y, and two processes, p_1, p_2. We omit the channel $\mathsf{e}(p_2, \mathsf{y})$ for space. Process p_1's frontiers are shown in violet, p_2's frontiers are shown in orange. We begin with the first memory configuration. The arrow depicts $\mathsf{constraint}(\mathsf{v}_1, \mathsf{y}, p_1) = \mathsf{v}_2$. This situation can arise in a causally consistent model where the writer of v_1 was aware of v_2 before writing v_1. The first transition shows p_2 updating and moving its frontier (to v_1). This results in a redundant node (v_3 in hashed texture) since the frontier of both p_1 and p_2 has crossed it. This is cleaned up by GC. Now, p_2 begins its read from v_1. Reading v_1, albeit on x, makes all writes by p_1 to y prior to v_2 redundant. When p_2 reads v_1, its frontier on $\mathsf{e}(\mathsf{y}, p_1)$ advances as prescribed by $\mathsf{constraint}(\mathsf{v}_1, \mathsf{y}, p_1)$, as shown in the fourth memory configuration. Note that this makes another message (v_4) redundant: all frontiers are past it. Once again, GC discards the message obtaining the last configuration.

3.3 Instantiating the Framework

Versatility of the framework The framework we introduce can be instantiated to RMO [12], FIFO consistency, RA, SRA, WRA [35,34], TSO [13], PSO, StrongCOH (the relaxed fragment of RC11) [30], and SC [40].
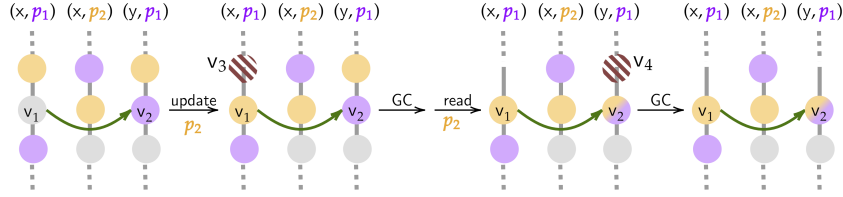
Fig. 8: Update, constraint in action during a read, and garbage collection

This claim is established by constructing semantics for each of these models using the components that we have discussed. We provide a summary of the insights, and defer the technical details to Appendix A.
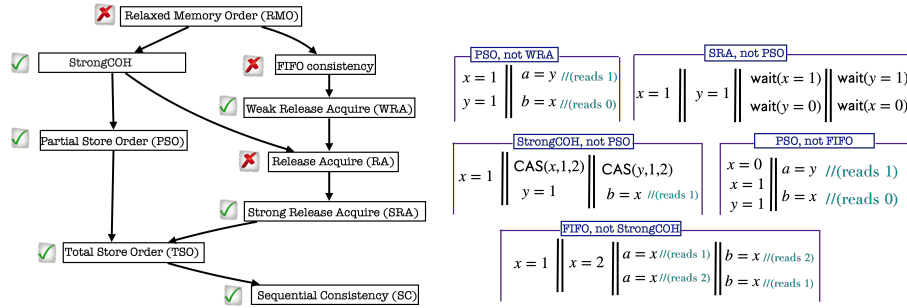
Fig. 9: Memory models, arranged by their strength. An arrow from A to B denotes that B is strictly more restrictive than A. A green check (resp. red cross) denotes the control state reachability is decidable (resp. undecidable).

We briefly explain how our framework accounts for the increasing restrictive strength of these memory models. The weakest of these is RMO, which only enforces poloc. There are no other causal dependencies, and thus for any message the constraint on other channels is $\top$. RMO can be strengthened in two ways: StrongCOH does it by requiring a total order on writes to the same variable, i.e. mo_x. Here the constraint is nontrivial only on channels of the same variable. On the other hand, FIFO enforces consistency with respect to the program order. Here, the constraint is nontrivial only on channels of the same process. WRA strengthens FIFO by enabling reads to enforce causal dependencies between write messages. This is captured by the non-trivial constraint, and we note that seen (the set of processes to have sourced a read from a message) plays a crucial role here. RA enforces the mo_x of StrongCOH as well as the causal dependencies of WRA. PSO strengthens StrongCOH by requiring a stronger precondition on the execution of an atomic read-write. More precisely, in any given configuration, for every variable, there is at most one write message that can be used to source a CAS operation, i.e. with the CAS flag set to true. SRA and TSO respectively

strengthen RA and PSO by doing away with write races. Here, the Boolean CAS flag in messages is all-important as an enforcer of atomicity. TSO strengthens SRA in the same way as PSO strengthens StrongCOH. Finally, when we get to SC, the model is so strong that all messages are instantly propagated. Here, for any message, the pointer on other channels is $\perp$.

4 Fairness properties

Towards the goal of verifying liveness, we use the framework we developed to introduce fairness properties in the the classical and probabilistic settings in §4.1 and §4.2 respectively. Our approach thus has the advantage of generalising over weak memory. In §4.3 we relate these fairness properties in the context of repeated control state reachability: a key liveness problem.

4.1 Transition and memory fairness

In this section, we consider fairness in the classical (non-probabilistic) case. We begin by defining transition fairness, which [10] is a standard notion of fairness that disallows executions which neglect certain transitions while only taking others. For utility in weak memory liveness, we then augment transition fairness to meet practical assumptions on the memory subsystem. Transition fairness and probabilistic fairness are intrinsically linked [27, Section 11]. Our augmentations are designed to carry over to the probabilistic domain in the same spirit.

Definition 1 (Transition fairness, [10]). *We say that a program execution is transition fair if for every configuration that is reached infinitely often, each transition enabled from it is also taken infinitely often.*

We argued the necessity of transition fairness in the introduction; however, it is vacuously satisfied by an execution that visits any configuration only finitely often. This could certainly be the case in weak memory, where there are infinitely many configurations. To make a case for the implausibility of this scenario, we begin by characterising classes of weak memory configurations.

Definition 2 (Configuration size). *Let γ be a program configuration with memory component* (e, frontier, source, constraint)*. We denote the configuration size by* size(γ) *and it is defined as* $\sum_x \sum_p$ len$(e(x, p))$*, i.e. the total number of messages in the message structure.*

Intuitively, the size of the configuration is the number of messages "in transit", and hence a measure of the weakness of the behaviour of the execution. We note that overly weak behaviour is rarely observed in practice [47,23]. For instance, instruction reorderings that could be observed as weak behaviour are limited in scope. Another source of weak behaviour is the actual reading of stale values at runtime. However, the hardware (i.e. caches, buffers, etc.) that stores these

values is finite, and is typically flushed regularly. Informally, the finite footprint of the system architecture (eg. micro-architecture) implies a bound, albeit hard to compute, on the size of the memory subsystem. Thus, we use the notion of configuration size to define:

Definition 3 (Size Bounded Executions). *An execution* $\gamma_0, \gamma_1, \ldots$ *is said to be size bounded, if there exists an N such that for all $n \in \mathbb{N}$, $\mathsf{size}(\gamma_n) \leq N$. If this N is specified, we refer to the execution as N-bounded.*

Already, the requirement of size-boundedness enables our system to refine our practical heuristics. However, if the bound N is unknown, it isn't immediately clear how this translates into a technique for liveness verification. We will now use the same rationale to motivate and develop an alternate augmentation which lends itself more naturally to algorithmic techniques. Recall that we intuitively relate the size of the configuration to the extent of weak behaviour. Now, consider Sequential Consistency, the strongest of the models. All messages are propagated immediately, and hence, the configuration has minimal size throughout. We call minimally sized configurations *plain*, and they are of particular interest to us:

Definition 4 (Plain message structure). *A message structure* $(\mathsf{V}, \mathsf{msgmap}, \mathsf{e})$ *is called plain, if for each variable* x, $\sum_p \mathsf{len}(\mathsf{e}(\mathsf{x}, p)) = 1$.

Drawing a parallel with SC, one could reason that the recurrence of plain configurations is a hallmark of a system that doesn't exhibit overly weak behaviour. This is captured with the following fairness condition.

Definition 5 (Repeatedly Plain Executions). *An execution* $\gamma_0, \gamma_1, \ldots$ *is said to be repeatedly plain, if* γ_i *is a plain configuration for infinitely many i.*

Following the memory transition system introduced in §2 and §3, we observe that every configuration has a (finite) path to some plain configuration (by performing a sequence of update steps). Hence, if a configuration is visited infinitely often in a fair execution, a plain configuration will also be visited infinitely often. Consequently, size bounded transition fair runs are also repeatedly plain transition fair.

4.2 Probabilistic memory fairness

Problems considered in a purely logical setting ask whether *all* executions satisfying a fairness condition fulfill a liveness requirement. However, if the answer is negative, one might be interested in quantifying the fair executions which do not repeatedly reach the control state. We perform this quantification by considering the probabilistic variant of the model proposed earlier, and defining fairness analogously as a property of Markov Chains.

Markov chains A Markov chain is a pair $\mathcal{C} = \langle \Gamma, \mathtt{M} \rangle$ where Γ is a (possibly-infinite) set of configurations and $\mathtt{M}$ is the transition matrix which assigns to each possible transition, a *transition probability*: $\mathtt{M} : \Gamma \times \Gamma \to [0,1]$. Indeed, this matrix needs to be stochastic, i.e., $\sum_{\gamma' \in \Gamma} \mathtt{M}(\gamma, \gamma') = 1$ should hold for all configurations.

We can convert our concurrent program transition (§2) into a Markov chain $\mathtt{M}$ by adding probabilities to the transitions. We assign $\mathtt{M}(\gamma, \gamma')$ to a nonzero value if and only if the transition $\gamma \to \gamma'$ is allowed in the underlying transition system. Markov Chain executions are, by construction, transition fair with probability 1. We now present the analog of the repeatedly plain condition. [5]

Definition 6 (Probabilistic Memory Fairness). *A Markov chain is considered to satisfy probabilistic memory fairness if a plain configuration is reached infinitely often with probability one.*

This parallel has immense utility because verifying liveness properties for a class of Markov Chains called Decisive Markov Chains is well studied. [6] establishes that the existence of a *finite attractor*, i.e a finite set of states F that is repeatedly reached with probability 1, is sufficient for decisiveness. The above definition asserts that the set of plain configurations is a finite attractor.

4.3 Relating fairness notions

Although repeatedly plain transition fairness is weaker than size bounded transition fairness and probabilistic memory fairness, these three notions are equivalent with respect to canonical liveness problems, i.e. repeated control state reachability and termination. The proof we present for repeated reachability can be adapted for termination.

Theorem 1. *There exists $N_0 \in \mathbb{N}$ such that for all $N \geq N_0$, the following are equivalent for any control state (program counters and register values) c:*

1. *All repeatedly plain transition fair runs visit c infinitely often.*

2. *All N-bounded transition fair runs visit c infinitely often.*

3. *c is visited infinitely often under probabilistic memory fairness with probability 1.*

Proof. For each $N \in \mathbb{N}$, we construct a connectivity graph $\mathcal{G}_N$. The vertices are the finitely many plain configurations γ, along with the finitely many control states c. We draw a directed edge from γ_i to γ_j, if γ_j is reachable from γ_i via configurations of size at most N. We additionally draw an edge from a plain configuration γ to control state c iff c is reachable from γ via configurations of

---

[5] A concrete Markov Chain satisfying the declarative definition may be adapted from the one described in [5] in a similar setting.

size at most N. We similarly construct a connectivity graph $\mathcal{G}$ without bounds on intermediate configuration sizes. We note:

1. There are only finitely many possibilities for $\mathcal{G}_N$

2. As N increases, edges can only be added to $\mathcal{G}_N$. This guarantees saturation.

3. Any witness of reachability is necessarily finite, hence the saturated graph is the same as $\mathcal{G}$, i.e. for all sufficiently large N, $\mathcal{G}_N = \mathcal{G}$

Since plain configurations are attractors, the graph $\mathcal{G}$ is instrumental in deciding repeated control state reachability. Consider the restriction of $\mathcal{G}$ to plain configurations, i.e. $\mathcal{G}_\Gamma$. Transition fairness (resp. Markov fairness) implies that γ is visited infinitely often (resp. with probability 1) only if it is in a bottom strongly connected component (scc). In turn any control state c will be guaranteed to be reached infinitely often if and only if it is reachable from every bottom scc of $\mathcal{G}_\Gamma$. The if direction follows using the transition fairness and attractor property, while the converse follows by simply identifying a finite path to a bottom scc from which c isn't reachable. The equivalence follows because the underlying graph is canonical for all three notions of fairness.

5 Applying fairness properties to decision problems

In this section, we show how to decide liveness as a corollary of the proof of Theorem 1. We begin by noting that techniques for termination are subsumed by those for repeated control state reachability. This is because termination is not guaranteed iff one can reach a plain configuration from which a terminal control state is inaccessible. Hence, in the sequel, we focus on repeated control state reachability.

5.1 Deciding repeated control state reachability

We observe that under the fairness conditions we defined, liveness, i.e. repeated control state reachability reduces to a *safety* query.

Problem 1 (Repeated control state reachability). Given a control state (program counters and register values) c, do all infinite executions (in the probabilistic case, a set of measure 1) satisfying fairness condition $\mathcal{A}$ reach c infinitely often?

Problem 2 (Control state reachability). Given a control state c and a configuration γ, is $(c, \gamma_\mathtt{m})$ reachable from γ for some $\gamma_\mathtt{m}$?

Theorem 2. *Problem 1 for repeatedly fair transition fairness and probabilistic memory fairness reduces to Problem 2. Moreover, the reduction can compute the N_0 from Theorem 1 such that it further applies to size bounded transition fairness.*

Proof. This follows by using Problem 2 to compute the underlying connectivity graph $\mathcal{G}$ from the proof of Theorem 1. A small technical hurdle is that plain configuration reachability is not the same as control state reachability. However, the key to encode this problem as a control state query is to use the following property: for a configuration γ and a message m $(\in \mathsf{e}(\mathsf{x}, p))$, if for every process p', m is not redundant (formally, $\mathsf{frontier}(p', \mathsf{x}, p) \leq m$), then there exists a plain configuration γ' containing m such that γ' is reachable from γ via a sequence of update steps. The plan, therefore, is to read and verify whether the messages desired in the plain configuration are, and remain accessible to all processes. Finally, the computation of N_0 follows by enumerating $\mathcal{G}_N$.

5.2 Quantitative Control State Repeated Reachability

We set the context of a Markov chain $\mathcal{C} = \langle \Gamma, \mathtt{M} \rangle$ that refines the underlying the transition system induced by the program. We consider is the quantitative variant of repeated reachability, where instead of just knowing whether the probability is one or not, we are interested in computing it.

Problem 3 (Quantitative control state repeated reachability). Given a control state c and an error margin $\epsilon \in \mathbb{R}$, find a δ such that for Markov chain $\mathcal{C}$, $|\mathrm{Prob}(\gamma_{\mathsf{init}} \models \Box\Diamond c) - \delta| \leq \epsilon$

We refer the reader to Appendix B for details on the standard reduction, from which the following result follows:

Theorem 3. *If Problem 2 is decidable for a memory model, then Problem 3 is computable for Markov chains that satsify probabilistic memory fairness.*

5.3 Adapting subroutines to our memory framework

We now briefly sketch how to adapt known solutions to Problem 2 for PSO, TSO, StrongCOH, WRA and SRA to our framework.

PSO and TSO Reachability between plain configurations (a special case of Problem 2) under these models has already been proven decidable [11]. The store buffer framework is similar to the one we describe, and hence the results go through. Moreover, [5, Lemmas 3, 4] shows the decidability of our Problem 2 for TSO. The same construction, which uses an augmented program to reduce to ex-plain configuration reachability, works for PSO as well.

StrongCOH Decidability of reachability under StrongCOH is shown in [1]. The framework used, although quite different in notation, is roughly isomorphic to the one we propose. The relaxed semantics of StrongCOH allow the framework to be set up as a WSTS [2,26], which supports backward reachability analysis,

yielding decidability. Backward reachability gives an upward closed set of states that can reach a target label. Checking whether an arbitrary state is in this upward closed set requires a comparison with only the finitely many elements in the basis. This solves Problems 2.

WRA and SRA Decidability of reachability under WRA and SRA has recently been shown in [34]. The proof follows the WSTS approach, however, the model used in the proof has different syntax and semantics from the one we present here. However, a reconciliation is possible, and we briefly sketch it here. A state in the proof model is a map from processes to *potentials*. A potential is a finite set of finite traces that a process may execute. These proof-model states are well-quasi-ordered and operating on them sets up a WSTS. Backward reachability gives us a set of maps from processes to *potentials* that allow us to reach the target label. The key is to view a process-potential map as a requirement on our message based configuration. Higher a map in the wqo, stronger the requirement it enforces. In this sense, the basis of states returned by backward reachability constitute the minimal requirements our configuration may meet in order for the target label to be reachable. Formally, let γ be a configuration of our framework. The target label is reachable from γ if and only if: there exists a process-potential map $\mathcal{B}$ is in the backward reachable set, such that every trace in every process' potential in $\mathcal{B}$ is enabled in γ. It suffices to check the existence of $\mathcal{B}$ over the finite basis of the backward reachable set. Note that γ is completely arbitrary: this solves our Problem 2.

6 Related Work

Fairness. Only recently has fairness for weak memory started receiving increasing attention. The work closest to ours is by [4], who formulate a probabilistic extension for the Total Store Order (TSO) memory model and show decidability results for associated verification problems. Our treatment of fairness is richer, as we relate same probabilistic fairness with two alternate logical fairness definitions. Similar proof techniques notwithstanding, our verification results are also more general, thanks to the development of a uniform framework that applies to a landscape of models. [37] develop a novel formulation of fairness as a declarative property of event structures. This notion informally translates to "Each message is eventually propagated." We forego axiomatic elegance to motivate and develop stronger practical notions of fairness in our quest to verify liveness.

Probabilistic Verification. There are several works on verification of *finite-state Markov chains* (e.g. [14,33]). However, since the messages in our memory systems are unbounded, these techniques do not apply. There is also substantive literature on the verification of infinite state probabilistic system, which have often been modelled as infinite Markov chains [18,25,24,17,19]. However their results cannot be directly leveraged to imply ours. The machinery we use for

showing decidability is relies on decisive Markov chains, a concept formulated in [6] and used in [4].

Framework. On the modelling front, the ability to specify memory model semantics as first-order constraints over the program-order (po), reads-from (rf), and modification-order (mo) have led to elegant declarative frameworks based on event structures [9,8,28,21]. There are also approaches that, instead of natively characterizing semantics, prescribe constraints on their ISA-level behaviours in terms of program transformations [38]. On the operational front, there have been works that model individual memory models [48,44] and clusters of similar model [35,30], however we are not aware of any operational modelling framework that encompasses as wide a range of models as we do. The operationalization in [16] uses write buffers which resemble our channels, however, their operationalization too focuses on a specific semantics.

7 Conclusion, Future Work, and Perspective

Conclusion The ideas developed in §4 lie at the heart of our contribution: we motivate and define transition fairness augmented with memory size boundedness or the recurrence of plain configurations, as well as the analogous probabilistic memory fairness. These are equivalent for the purpose of verifying repeated control state reachability, i.e. liveness, and lie at the core of the techniques we discuss in §5. These techniques owe their generality to the versatile framework we describe in §3.

Future Work There are several interesting directions for future work. We believe that our framework can be extended to handle weak memory models that allow speculation, such as ARM and POWER. In such a case, we would need to extend our fairness conditions to limit the amount of allowed speculation. It is also interesting to mix transition fairness with probabilistic fairness, i.e., use the former to solve scheduler non-determinism and the latter to resolve memory non-determinism, leading to (infinite-state) Markov Decision Process model. Along these lines, we can also consider synthesis problems based on $2\frac{1}{2}$-games. To solve such game problems, we could extend the framework of Decisive Markov chains that have been developed for probabilistic and game theoretic problems over infinite-state systems [6] A natural next step is developing efficient algorithms for proving liveness properties for programs running on weak memory models. In particular, since we reduce the verification of liveness properties to simple reachability, there is high hope one can develop CEGAR frameworks relying both on over-approximations, such as predicate abstraction, and under-approximations such as bounded context-switching [46] and stateless model checking [32,3].

Perspective Leveraging techniques developed over the years by the program verification community, and using them to solve research problems in programming languages, architectures, databases, etc., has substantial potential added value.

Although it requires a deep understanding of program behaviors running on such platforms, we believe it is about finding the right concepts, combining them correctly, and then applying the existing rich set of program verification techniques, albeit in a non-trivial manner. The current paper is a case in point. Here, we have used a combination of techniques developed for reactive systems [31], methods for the analysis of infinite-state systems [6], and semantical models developed for weak memory models [35,34,12,30] to obtain, for the first time, a framework for the systematic analysis of liveness properties under weak memory models.

References

1. Abdulla, P., Atig, M.F., Godbole, A., Krishna, S., Vafeiadis, V.: The Decidability of Verification under PS 2.0, pp. 1–29 (03 2021). https://doi.org/10.1007/978-3-030-72019-3_1
2. Abdulla, P.A.: Well (and better) quasi-ordered transition systems. Bull. Symb. Log. **16**(4), 457–515 (2010). https://doi.org/10.2178/bsl/1294171129, https://doi.org/10.2178/bsl/1294171129
3. Abdulla, P.A., Aronis, S., Jonsson, B., Sagonas, K.: Source sets: A foundation for optimal dynamic partial order reduction. J. ACM **64**(4), 25:1–25:49 (2017)
4. Abdulla, P.A., Atig, M.F., Agarwal, R.A., Godbole, A., Narayanan, K.S.: Probabilistic total store ordering. In: ESOP (2022)
5. Abdulla, P.A., Atig, M.F., Agarwal, R.A., Godbole, A., S, K.: Probabilistic total store ordering (2022). https://doi.org/10.48550/ARXIV.2201.10213, https://arxiv.org/abs/2201.10213
6. Abdulla, P.A., Henda, N.B., Mayr, R.: Decisive markov chains. LMCS **3**(4) (2007)
7. Adve, S.V., Gharachorloo, K.: Shared memory consistency models: A tutorial. IEEE Computer **29**(12), 66–76 (1996)
8. Alglave, J.: A formal hierarchy of weak memory models. Formal Methods in System Design **41**, 178–210 (2012)
9. Alglave, J., Maranget, L., Tautschnig, M.: Herding cats: modelling, simulation, testing, and data-mining for weak memory. Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (2014)
10. Aminof, B., Ball, T., Kupferman, O.: Reasoning about systems with transition fairness. In: Baader, F., Voronkov, A. (eds.) Logic for Programming, Artificial Intelligence, and Reasoning. pp. 194–208. Springer Berlin Heidelberg, Berlin, Heidelberg (2005)
11. Atig, M.F., Bouajjani, A., Burckhardt, S., Musuvathi, M.: On the verification problem for weak memory models. In: POPL (2010)
12. Atig, M.F., Bouajjani, A., Burckhardt, S., Musuvathi, M.: On the verification problem for weak memory models. In: Hermenegildo, M.V., Palsberg, J. (eds.) Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010. pp. 7–18. ACM (2010). https://doi.org/10.1145/1706299.1706303, https://doi.org/10.1145/1706299.1706303
13. Atig, M.F., Bouajjani, A., Parlato, G.: Getting Rid of Store-Buffers in TSO Analysis. In: CAV. LNCS, vol. 6806, pp. 99–115. Springer (2011)
14. Baier, C., Katoen, J.P.: Principles of Model Checking (Representation and Mind Series). The MIT Press (2008)

15. Batty, M., Owens, S., Sarkar, S., Sewell, P., Weber, T.: Mathematizing c++ concurrency. In: POPL. pp. 55–66. ACM (2011)
16. Boudol, G., Petri, G.: Relaxed memory models: an operational approach. In: POPL '09 (2009)
17. Brázdil, T., Chatterjee, K., Kucera, A., Novotný, P., Velan, D.: Deciding fast termination for probabilistic VASS with nondeterminism. In: Chen, Y., Cheng, C., Esparza, J. (eds.) Automated Technology for Verification and Analysis - 17th International Symposium, ATVA 2019, Taipei, Taiwan, October 28-31, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11781, pp. 462–478. Springer (2019). https://doi.org/10.1007/978-3-030-31784-3_27, https://doi.org/10.1007/978-3-030-31784-3_27
18. Brázdil, T., Kiefer, S., Kucera, A., Novotný, P., Katoen, J.: Zero-reachability in probabilistic multi-counter automata. In: Henzinger, T.A., Miller, D. (eds.) Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS '14, Vienna, Austria, July 14 - 18, 2014. pp. 22:1–22:10. ACM (2014). https://doi.org/10.1145/2603088.2603161, https://doi.org/10.1145/2603088.2603161
19. Brázdil, T., Kiefer, S., Kucera, A., Vareková, I.H.: Runtime analysis of probabilistic programs with unbounded recursion. J. Comput. Syst. Sci. **81**(1), 288–310 (2015). https://doi.org/10.1016/j.jcss.2014.06.005, https://doi.org/10.1016/j.jcss.2014.06.005
20. Broy, M., Wirsing, M.: On the algebraic specification of nondeterministic programming languages. In: CAAP (1981)
21. Chakraborty, S.S., Vafeiadis, V.: Grounding thin-air reads with event structures. Proceedings of the ACM on Programming Languages **3**, 1 – 28 (2019)
22. Doherty, S., Dongol, B., Wehrheim, H., Derrick, J.: Verifying C11 programs operationally. In: Hollingsworth, J.K., Keidar, I. (eds.) Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2019, Washington, DC, USA, February 16-20, 2019. pp. 355–365. ACM (2019). https://doi.org/10.1145/3293883.3295702, https://doi.org/10.1145/3293883.3295702
23. Elver, M., Nagarajan, V.: TSO-CC: consistency directed cache coherence for TSO. In: HPCA 2014. pp. 165–176. IEEE (2014)
24. Esparza, J., Kucera, A., Mayr, R.: Model checking probabilistic pushdown automata. In: 19th IEEE Symposium on Logic in Computer Science (LICS 2004), 14-17 July 2004, Turku, Finland, Proceedings. pp. 12–21. IEEE Computer Society (2004). https://doi.org/10.1109/LICS.2004.1319596, https://doi.org/10.1109/LICS.2004.1319596
25. Etessami, K., Yannakakis, M.: Recursive markov decision processes and recursive stochastic games. J. ACM **62**(2), 11:1–11:69 (2015). https://doi.org/10.1145/2699431, https://doi.org/10.1145/2699431
26. Finkel, A., Schnoebelen, P.: Well-structured transition systems everywhere! Theor. Comput. Sci. **256**(1-2), 63–92 (2001). https://doi.org/10.1016/S0304-3975(00)00102-X, https://doi.org/10.1016/S0304-3975(00)00102-X
27. van Glabbeek, R.J., Höfner, P.: Progress, justness, and fairness. ACM Computing Surveys (CSUR) **52**, 1 – 38 (2019)
28. Jeffrey, A., Riely, J.: On thin air reads towards an event structures model of relaxed memory. 2016 31st Annual ACM/IEEE Symposium on Logic in Computer Science (LICS) pp. 1–9 (2016)

29. Kaiser, J., Dang, H., Dreyer, D., Lahav, O., Vafeiadis, V.: Strong logic for weak memory: Reasoning about release-acquire consistency in iris. In: Proceedings of the 31st European Conference on Object-Oriented Programming, ECOOP 2017. pp. 17:1–17:29 (2017)

30. Kang, J., Hur, C., Lahav, O., Vafeiadis, V., Dreyer, D.: A promising semantics for relaxed-memory concurrency. In: POPL 2017. pp. 175–189 (2017)

31. Kesten, Y., Pnueli, A., Raviv, L., Shahar, E.: Model checking with strong fairness. Formal Methods Syst. Des. **28**(1), 57–84 (2006). https://doi.org/10.1007/s10703-006-4342-y, https://doi.org/10.1007/s10703-006-4342-y

32. Kokologiannakis, M., Lahav, O., Sagonas, K., Vafeiadis, V.: Effective stateless model checking for C/C++ concurrency. PACMPL **2**, 17:1–17:32 (2018)

33. Kwiatkowska, M.Z., Norman, G., Parker, D.: PRISM 4.0: Verification of probabilistic real-time systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6806, pp. 585–591. Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_47, https://doi.org/10.1007/978-3-642-22110-1_47

34. Lahav, O., Boker, U.: What's decidable about causally consistent shared memory? ACM Trans. Program. Lang. Syst. **44**(2) (apr 2022). https://doi.org/10.1145/3505273, https://doi.org/10.1145/3505273

35. Lahav, O., Giannarakis, N., Vafeiadis, V.: Taming release-acquire consistency. In: Bodík, R., Majumdar, R. (eds.) Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016. pp. 649–662. ACM (2016)

36. Lahav, O., Namakonov, E., Oberhauser, J., Podkopaev, A., Vafeiadis, V.: Making weak memory models fair. ArXiv **abs/2012.01067** (2020)

37. Lahav, O., Namakonov, E., Oberhauser, J., Podkopaev, A., Vafeiadis, V.: Making weak memory models fair (2020). https://doi.org/10.48550/ARXIV.2012.01067, https://arxiv.org/abs/2012.01067

38. Lahav, O., Vafeiadis, V.: Explaining relaxed memory models with program transformations. In: FM (2016)

39. Lahav, O., Vafeiadis, V., Kang, J., Hur, C.K., Dreyer, D.: Repairing sequential consistency in c/c++11. Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (2017)

40. Lamport, L.: How to make a multiprocessor that correctly executes multiprocess programs. IEEE Trans. on Computers **C-28**, 690–691 (1979)

41. Mador-Haim, S., Maranget, L., Sarkar, S., Memarian, K., Alglave, J., Owens, S., Alur, R., Martin, M.M.K., Sewell, P., Williams, D.: An axiomatic memory model for power multiprocessors. In: CAV (2012)

42. Manna, Z., Pnueli, A.: The temporal logic of reactive and concurrent systems - specification. Springer (1992). https://doi.org/10.1007/978-1-4612-0931-7, https://doi.org/10.1007/978-1-4612-0931-7

43. McKinley, K.S., Fisher, K. (eds.): Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019. ACM (2019). https://doi.org/10.1145/3314221, https://doi.org/10.1145/3314221

44. Nienhuis, K., Memarian, K., Sewell, P.: An operational semantics for c/c++11 concurrency. In: OOPSLA (2016)

45. Owens, S., Sarkar, S., Sewell, P.: A better x86 memory model: x86-tso. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) Theorem Proving in Higher

Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5674, pp. 391–407. Springer (2009)

46. Qadeer, S., Rehof, J.: Context-bounded model checking of concurrent software. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS. Lecture Notes in Computer Science, vol. 3440, pp. 93–107. Springer (2005). https://doi.org/10.1007/978-3-540-31980-1_7, https://doi.org/10.1007/978-3-540-31980-1_7

47. Ros, A., Kaxiras, S.: Callback: Efficient synchronization without invalidation with a directory just for spin-waiting. In: ISCA. pp. 427–438 (2015)

48. Sewell, P., Sarkar, S., Owens, S., Nardelli, F.Z., Myreen, M.O.: x86-tso: a rigorous and usable programmer's model for x86 multiprocessors. Commun. ACM **53**(7), 89–97 (2010)

49. Wolper, P.: Expressing interesting properties of programs in propositional temporal logic. In: POPL. pp. 184–193. ACM Press (1986). https://doi.org/10.1145/512644.512661, http://doi.acm.org/10.1145/512644.512661

A Instantiations of our framework

We now provide the technical justification of the versatility of the framework, as stated in §3.3.

A.1 Instantiating RMO, FIFO, WRA

First we describe a write operation to x by a process p, resulting in the addition of a new message node v_{new}. We append v_{new} with a `true` CAS indicator to the end of $e(x, p)$. In RMO, the only coherence requirement is on writes by the same process to the same variable. Thus, on reading v_{new}, the frontiers channels other than $e(x, p)$ need not advance. FIFO strengthens RMO by requiring coherence across writes issued by the same process. Here, on reading v_{new}, the frontiers of channels $e(y, p)$ with $y \neq x$ (of other variables) need not advance. WRA strengthens FIFO by requiring that the causal order among events may not be irreflexive. Here, reading v_{new} imposes constraints on frontiers in all channels.

Example 3. We shall now briefly illustrate WRA, the most complex of the three models, with a small run. Again, we have two processes p_1, p_2, whose frontiers are shown in blue and red respectively. We show constraints only when they are in action. The shared memory consists of variables x, y. Let p_1 make write v_1 to x. Now, p_2 makes a write v_2 to x, reads v_1, and finally writes v_3 to x. At this point, assuming no updates that non-deterministically advance them, p_1's frontiers will be at v_1 and v_2. p_2's frontier on $e(x, p_2)$ is obviously at v_3. However, v_3, having been written *after reading* v_1, is causally after v_1. Hence, the frontier of p_2 on $e(x, p_1)$ is $\perp$: i.e. it considers v_1 to be in the past, or redundant. This is illustrated in the figure as the red pointer at the end of the channel.
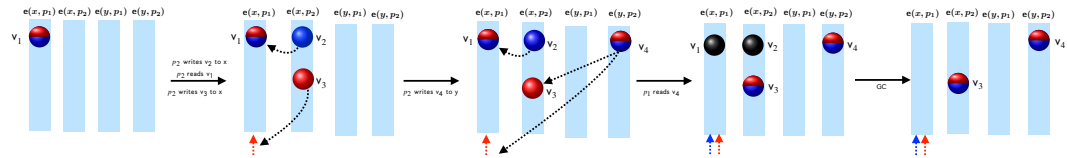
Here, $\mathsf{constraint}(v_3, x, p_1) = \perp$, corroborating this. p_2 now writes v_4 to y. As per the frontier of p_2, and causal consistency, reading v_4 must render v_1 and v_2 redundant. This is captured by having the constraint point to $\perp$ and v_3 respectively. Finally, assume p_1 reads from v_4. As per the constraint, its frontiers advance to $\perp$ in $e(x, p_1)$ and v_3 in $e(x, p_2)$. Each frontier has now moved past v_1 and v_2, and garbage collection discards them, when called upon.

Writes When a write is made by a process p on variable x, we assimilate the new message with the following changes to the memory configuration, performed sequentially:

write RMO, FIFO, WRA

$$\mathsf{v}_{new}.\mathsf{value} = \mathsf{d}$$

$$\mathsf{v}_{new}.\mathsf{CAS} = \mathtt{true} \qquad \gamma_1 = \gamma_0[\mathsf{e}(\mathsf{x}, p) \leftarrow \mathsf{e}(\mathsf{x}, p) \cdot \mathsf{v}_{new}][\mathsf{frontier}(p, \mathsf{x}, p) \leftarrow \mathsf{v}_{new}]$$

$$(\text{WRA}) \; \gamma_2 = \gamma_1[\lambda p' \neq p. \;\; \mathsf{frontier}(p, \mathsf{x}, p') \leftarrow \mathsf{frontier}(p, \mathsf{x}, p').\mathsf{nextmsg} \; \text{if } p \in \mathsf{frontier}(p, \mathsf{x}, p').\mathsf{seen}]$$

$$(\text{RMO, FIFO}) \; \gamma_2 = \gamma_1 \qquad \gamma_3 = \gamma_2[\lambda \mathsf{y}.\lambda p'.\mathsf{constraint}(\mathsf{v}_{new}, \mathsf{y}, p') \leftarrow \top]$$

$$(\text{RMO}) \; \gamma_4 = \gamma_3[\mathsf{constraint}(\mathsf{v}_{new}, \mathsf{x}, p) \leftarrow \mathsf{frontier}(p, \mathsf{x}, p)]$$

$$(\text{FIFO}) \; \gamma_4 = \gamma_3[\lambda \mathsf{y}. \;\; \mathsf{constraint}(\mathsf{v}_{new}, \mathsf{y}, p) \leftarrow \mathsf{frontier}(p, \mathsf{y}, p)]$$

$$(\text{WRA}) \; \gamma_4 = \gamma_3[\lambda \mathsf{y}.\lambda p'.\mathsf{constraint}(\mathsf{v}_{new}, \mathsf{y}, p') \leftarrow \mathsf{frontier}(p, \mathsf{y}, p')]$$

$$\gamma_5 = \gamma_4[\lambda p'.\mathsf{frontier}(p', \mathsf{x}, p) \leftarrow \mathsf{v}_{new} \; \text{if } \mathsf{frontier}(p', \mathsf{x}, p) = \bot \text{ or } \epsilon][\lambda \mathsf{v}. \;\; \mathsf{constraint}(\mathsf{v}, \mathsf{x}, p) \leftarrow \mathsf{v}_{new} \; \text{if } \mathsf{constraint}(\mathsf{v}, \mathsf{x}, p) = \bot \text{ or } \epsilon]$$

$$\gamma_6 = \mathsf{GC}(\gamma_5)$$

$$\overline{\gamma_0 \xrightarrow{p:\mathtt{W(x,d)}}_{\mathtt{m}} \gamma_6}$$

update RMO, FIFO, WRA

$$\mathsf{v}_0 \in \gamma_0.\mathsf{e}(\mathsf{x}, p_0) \qquad \forall p'. \;\; \gamma_0.\mathsf{frontier}(p', \mathsf{x}, p_0) \leq \mathsf{v}_0 \qquad \mathsf{v}_1 \in \gamma_0.\mathsf{e}(\mathsf{x}, p_1) \qquad \mathsf{v}_1 \neq \mathsf{v}_0$$

$$\mathsf{v}_1 = \gamma_0.\mathsf{frontier}(p, \mathsf{x}, p_1)$$

$$\gamma_1 = \gamma_0[\mathsf{frontier}(p, \mathsf{x}, p_1) \leftarrow \mathsf{v}_1.\mathsf{nextmsg}][\mathsf{source}(p_2, \mathsf{y}) \leftarrow p_3] \qquad \gamma_2 = \mathsf{GC}(\gamma_1)$$

$$\overline{\gamma_0 \xrightarrow{\mathtt{m:Upd}}_{\mathtt{m}} \gamma_2}$$

CAS-fail RMO, FIFO, WRA

$$p_s = \gamma_0.\mathsf{source}(p, \mathsf{x}) \qquad \mathsf{v} = \gamma_0.\mathsf{frontier}(p, \mathsf{x}, p_s)$$

$$\mathsf{v}.\mathsf{value} = \mathsf{d}'' \neq \mathsf{d} \qquad \mathsf{v}.\mathsf{CAS} = \mathtt{true}$$

$$\cfrac{\gamma_0 \xrightarrow{p:\mathtt{R(x,d'')}}_{\mathtt{m}} \gamma_1}{\gamma_0 \xrightarrow{p:\mathtt{CAS(x,d,d',F)}}_{\mathtt{m}} \gamma_1}$$

CAS-succeed RMO, FIFO, WRA

$$p_s = \gamma_0.\mathsf{source}(p, \mathsf{x}) \qquad \mathsf{v} = \gamma_0.\mathsf{frontier}(p, \mathsf{x}, p_s)$$

$$\mathsf{v}.\mathsf{value} = \mathsf{d} \qquad \mathsf{v}.\mathsf{CAS} = \mathtt{true}$$

$$\gamma_0 \xrightarrow{p:\mathtt{R(x,d)}}_{\mathtt{m}} \gamma_1$$

$$\gamma_2 = \gamma_1[\mathsf{v}.\mathsf{CAS} \leftarrow \mathtt{false}] \qquad \gamma_2 \xrightarrow{p:\mathtt{W(x,d)}}_{\mathtt{m}} \gamma_3$$

$$\overline{\gamma_0 \xrightarrow{p:\mathtt{CAS(x,d,d',T)}}_{\mathtt{m}} \gamma_3}$$

Fig. 10: Transitions for RMO, FIFO, WRA

▷ The frontier is advanced to v_{new}: $\mathsf{frontier}(p, \mathsf{x}, p) \leftarrow \mathsf{v}_{new}$. Additionally to respect causality in WRA, we ensure that all messages used by p are lower than v_{new}. That is if $\mathsf{frontier}(p, \mathsf{x}, p')$ has been used by p, then $\mathsf{frontier}(p, \mathsf{x}, p') \leftarrow \mathsf{frontier}(p, \mathsf{x}, p').\mathsf{nextmsg}$.

▷ In all three models, $\mathsf{constraint}(\mathsf{v}_{new}, \mathsf{x}, p) \leftarrow \mathsf{v}_{new}$

▷ The constraints on $\mathsf{frontier}$ after reading v_{new} differ in the three models: In RMO, for all $(\mathsf{y}, p') \neq (\mathsf{x}, p)$ $\mathsf{constraint}(\mathsf{v}_{new}, \mathsf{y}, p') \leftarrow \top$ (ϵ if channel empty). This is to reflect that frontiers advance independently in all channels, there is no notion of causality across channels. In FIFO, for all $p' \neq p$, and all y, $\mathsf{constraint}(\mathsf{v}_{new}, \mathsf{y}, p') \leftarrow \top$ (ϵ if channel empty). However, for all y, $\mathsf{constraint}(\mathsf{v}_{new}, \mathsf{y}, p) \leftarrow \mathsf{frontier}(p, \mathsf{y}, p)$. This maintains a causal order among writes by the same process, which must be respected by reads. However, writes by different processes are not causally related. In WRA, for all y, p', $\mathsf{constraint}(\mathsf{v}_{new}, \mathsf{y}, p') \leftarrow \mathsf{frontier}(p, \mathsf{y}, p')$. Here, FIFO is strengthened by causally relating writes, with both the program order as in FIFO, and the reads-from relation. That is, reads are used to synchronise frontiers across processes and variables.

▷ For all other threads p', if $\mathsf{frontier}(p', \mathsf{x}, p) = \epsilon$ or $\bot$, then we update $\mathsf{frontier}(p', \mathsf{x}, p) \leftarrow \mathsf{v}_{new}$ i.e. if the channel was empty, or the frontier considered all existing messages in the past, then we update the frontier to consider our new message as the present

▷ Similarly, we correct the other type of reference and ensure it doesn't remain overly strong in view of the new message: for all other v, if $\mathsf{constraint}(\mathsf{v}, \mathsf{x}, p) = \epsilon$ or $\bot$: $\mathsf{constraint}(\mathsf{v}, \mathsf{x}, p) \leftarrow \mathsf{v}_{new}$. Finally, we run garbage collection.

Update The update step is uniform across the models: We identify channel $e(x, p_0)$ having message v_0 s.t. for all p', $\mathsf{frontier}(p', x, p_0) \leq v_0$. This will always exist, because in an operational semantics, the last write to x satisfies this property. We also identify a v_1, the frontier for some process p, and advance the frontier of p from v_1 to the next message. We then non-deterministically change $\mathsf{source}(p', x)$ for some p'. Finally, we run garbage collection and return.

CAS A CAS is straightforward: we check that the read source has the Boolean CAS indicator enabled, and read from it. If the read matches the expected value, then we further set the indicator to false, and make the write as usual.

A.2 Instantiating StrongCOH, RA, SRA

We now consider StrongCOH, RA, SRA which are fragments of the C11 [44] standard. Alternative opertional and axiomatic models for these have been proposed (e.g. [43,35]).

Per-variable coherence orders $\{\mathsf{mo_x}\}_x$. The property that connects Strong-COH, RA, SRA is that these models enforce a total per-variable coherence order (commonly denoted as the modification order - $\mathsf{mo_x}$). This is the order in which messages on the same variable are propagated to every process. A total $\mathsf{mo_x}$ strengthens the per-variable-per-process channels from our framework: $\mathsf{mo_x}$ is a linearization of the per-process channels $\{e(x, p)\}_p$. That is, at any given point, any reading process p can only read from one message from the channels $\{e(x, p')\}_{p'}$. We enforce this by ensuring that $\mathsf{frontier}(p, x, p') \in \mathbb{N}$ for exactly one p'. We call this unique message $\mathsf{frontier}'(p, x)$. For all other processes p'', it is an internode, i.e. $\mathsf{frontier}(p, x, p'') \in \mathbb{N}^+$. Timestamp-based operational models [35] use the integer-ordered timestamps to encode this ordering, while in our case, the channel orders along with the $\mathsf{constraint}$ state encodes the order. We enforce totality of this order as an invariant over e-chain and $\mathsf{constraint}$ orders, which we discuss through an example.

	$e(x, p_1)$	$e(x, p_2)$	$e(x, p_3)$
p_4	v_1	$\top$	$\top$
p_5	$\bot$	v_2	$\top$
p_6	$\bot$	v_2^+	v_4

frontier

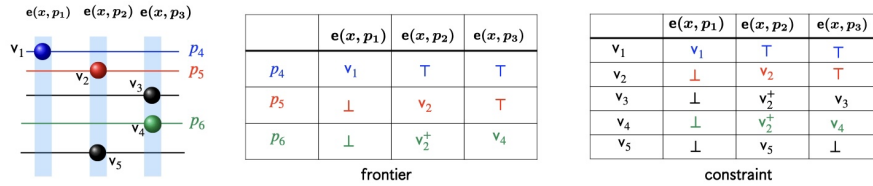|	$e(x, p_1)$	$e(x, p_2)$	$e(x, p_3)$
v_1	v_1	$\top$	$\top$
v_2	$\bot$	v_2	$\top$
v_3	$\bot$	v_2^+	v_3
v_4	$\bot$	v_2^+	v_4
v_5	$\bot$	v_5	$\bot$

constraint

Fig. 11: Maintenance of per-variable total order

The picture shows the total order on variable x. Nodes towards the bottom represent messages later in the total order. $\mathsf{frontier}'(p_4, x) = v_1$, $\mathsf{frontier}'(p_5, x) = v_2$, $\mathsf{frontier}'(p_6, x) = v_4$. As we can see in the frontier table, the values are already

indicative of the total order. The key idea in the maintenance of total order using constraint is to use the semantics: in particular, if process p reads message v, then for all p', $\mathsf{frontier}(p, \mathsf{x}, p')$, must be at least as far ahead as $\mathsf{constraint}(\mathsf{v}, \mathsf{x}, p')$. So, for a message v in $\mathsf{e}(\mathsf{x}, p)$, we define $\mathsf{constraint}(\mathsf{v}, \mathsf{x}, p) = \mathsf{v}$. Now consider the total order: in every other channel $\mathsf{e}(\mathsf{x}, p')$, let $\mathsf{v}_{p'}$ be the last message earlier than v in the total order. We set $\mathsf{constraint}(\mathsf{v}, \mathsf{x}, p') = \mathsf{v}_{p'}^{+}$.

StrongCOH $\supseteq$ RA $\supseteq$ SRA. RA strengthens Strong-COH by enforcing irreflexivity of per-variable RA order [36]. In particular, when p reads a message, p learns about all causally (hb)-previous messages to the read message. This is exemplified through the MP litmus

```
x=1  // v⁰ₓ ‖ r=y  // 1
y=1  // v⁰ᵧ ‖ s=x  // 0
```

Fig. 12: MP

test Fig. 12, where the annotated outcome is possible in StrongCOH but not in RA. We enforce this by requiring that processes update frontier for all variables during reads. The update is guided by the constraint state which, for every message, records the messages that are causally-previous to it. We account for the relaxation in StrongCOH by using differing semantics for the write operation (Ⓐ in Fig. 14).

SRA strengthens RA by eliminating write-write races, which is exemplified through the which we now discuss through Figure 13. While RA allows (hb $\cup$ mo)-cycles, SRA does not; SRA strengthens RA by forcing the incoming write to be placed last

```
x=1  // v⁰ₓ ‖ y=1  // v¹ᵧ
y=2  // v⁰ᵧ ‖ x=2  // v¹ₓ
r=y  // 1  ‖ s=x  // 1
```

Fig. 13: 2+2W

in the global coherence order. The outcome annotated in Fig. 13, requires both writes with value 2 to be arbitrated before those with value 1. This is allowed under RA but not under SRA. We enforce this by requiring that in SRA (Ⓑ in Fig. 14) the written message immediately follows its predecessor, i.e. the CAS flag on the predecessor is set to false blocking any intervening writes in the future.

Writes We now elaborate on the semantics of a write transition (Fig. 14). A write step by process p on variable x constructs a new message node v_{new}. The CAS flag of this message is set to true to indicate that there is no message on the e-chain immediately after v_{new}. The message v_{new} is inserted as the immediate successor of $\mathsf{v}_{prev} = \mathsf{frontier}'(p, \mathsf{x})$. Then we append the newly written message v_{new} to the chain $\mathsf{e}(\mathsf{x}, p)$ and $\mathsf{frontier}(p, \mathsf{x}, \_)$ is updated to v_{new}. Next, we update $\mathsf{constraint}(\mathsf{v}_{new}, \mathsf{y}, p')$ for all other variables and processes (Ⓐ). This is where the models differ. In RA/SRA the new message sees all messages know to p as its causal predecessors, while in the case of StrongCOH, the message only sees the messages on the written variable x. Hence, we define this as a model dependent update step with the functions: $\mathsf{f}_{\mathrm{StrongCOH}} = [\lambda p'.\lambda \mathsf{y}.\ \text{if } \mathsf{y} = \mathsf{x}\ \mathsf{frontier}(p, \mathsf{x}, p')\ \text{else } \top]$ and $\mathsf{f}_{\mathrm{RA,\ SRA}} = [\lambda p'.\lambda \mathsf{y}.\mathsf{frontier}(p, \mathsf{y}, p')]$.

Finally we update frontier for all other processes and constraint for all other messages to reflect the fact that v_{new} is being inserted after v_{prev}. This involves

$$\mathsf{v}_{new}.\mathsf{value} = \mathsf{d} \qquad \mathsf{v}_{new}.\mathsf{CAS} = \mathtt{true}$$

$$\mathsf{v}_{prev} = \gamma_0.\mathsf{frontier}'(p, \mathsf{x}) \in \mathbb{V} \qquad \mathsf{v}_{prev}.\mathsf{CAS} = \mathtt{true}$$

$$\gamma_1 = \gamma_0[\mathsf{e}(\mathsf{x}, p) \leftarrow \mathsf{e}(\mathsf{x}, p) \cdot \mathsf{v}_{new}][\mathsf{frontier}(p, \mathsf{x}, p) \leftarrow \mathsf{v}_{new}]$$

$$\gamma_2 = \gamma_1[\mathsf{frontier}(p, \mathsf{x}, p_s) \leftarrow \mathsf{frontier}(p, \mathsf{x}, p_s).\mathsf{nextpos} \text{ if } p \neq p_s]$$

$$\textcircled{A} \quad \gamma_3 = \gamma_2[\lambda\mathsf{y}.\lambda p'.\mathsf{constraint}(\mathsf{v}_{new}, \mathsf{y}, p') \leftarrow \mathsf{f}_{\mathrm{model}}]$$

$$\gamma_4 = insert(\gamma_3, \mathsf{v}_{new})$$

$$\textcircled{B} \quad \gamma_5 = \gamma_4[\mathsf{v}_{prev}.\mathsf{CAS} \leftarrow \mathsf{c}_{\mathrm{model}}] \qquad \gamma_6 = \mathsf{GC}(\gamma_5)$$

$$\gamma_0 \xrightarrow{\;p:\mathtt{W(x,d)}\;}_{\mathtt{m}} \gamma_6$$

Fig. 14: **write** transition semantics for StrongCOH, RA, SRA

the shifting these pointers by one place along the chain-positions discussed earlier (and accounting for some corner cases). We denote this as the $insert(\gamma, \mathsf{v}_{new})$ operation.

Finally, we set the CAS flag of v_{prev} to false in the case of SRA. This is again captured using a model-dependent rule (\textcircled{B}): $\mathsf{c}_{StrongCOH,RA} = \mathtt{true}$ and $\mathsf{c}_{SRA} = \mathtt{false}$. In the case of SRA, setting the CAS flag to false prevents future insertion of a new message in a non-maximal position. We conclude with garbage collection.

$$\mathsf{v}_0 \in \gamma_0.\mathsf{e}(\mathsf{x}, p_0) \qquad \forall p'. \; \gamma_0.\mathsf{frontier}(p', \mathsf{x}, p_0) \leq \mathsf{v}_0 \qquad \mathsf{v}_1 \in \gamma_0.\mathsf{e}(\mathsf{x}, p_1) \qquad \mathsf{v}_1 \neq \mathsf{v}_0$$

$$\mathsf{v}_1 = \gamma_0.\mathsf{frontier}(p, \mathsf{x}, p_1)$$

$$\mathsf{v}_2 \in \gamma_0.\mathsf{e}(\mathsf{x}, p_1) \qquad \gamma_0.\mathsf{constraint}(\mathsf{v}_1, \mathsf{x}, p_2) < \mathsf{v}_2 \qquad \gamma_0.\mathsf{constraint}(\mathsf{v}_2, \mathsf{x}, p_1) > \mathsf{v}_1$$

$$\forall p' \neq p_1, p_2. \; \gamma_0.\mathsf{constraint}(\mathsf{v}_1, \mathsf{x}, p') = \gamma_0.\mathsf{constraint}(\mathsf{v}_2, \mathsf{x}, p')$$

$$\gamma_1 = \gamma_0[\mathsf{frontier}(p, \mathsf{x}, p_1) \leftarrow \mathsf{v}_1.\mathsf{nextpos}][\mathsf{frontier}(p, \mathsf{x}, p_2) \leftarrow \mathsf{v}_2][\mathsf{source}(p_3, \mathsf{y}) \leftarrow p_4]$$

$$\gamma_2 = \mathsf{GC}(\gamma_1)$$

$$\gamma_0 \xrightarrow{\;\mathtt{m:Upd}\;}_{\mathtt{m}} \gamma_2$$

Fig. 15: **update** transition semantics for StrongCOH, RA, SRA

Update The update step revolves around changing $\mathsf{frontier}'(p, \mathsf{x})$ to its immediate coherence order successor, if defined. Let $\mathsf{v}_1 = \mathsf{frontier}'(p, \mathsf{x})$, and let write message v_2 be its successor in the total coherence order. We assume that v_1 and v_2 were written by p_1 and p_2 respectively. The update step: $\mathsf{frontier}(p, \mathsf{x}, p_1) \leftarrow \mathsf{v}_1.\mathsf{nextpos} \; \mathsf{frontier}(p, \mathsf{x}, p_2) \leftarrow \mathsf{v}_2$.

CAS A CAS is an atomic combination of a read and a write. When we read, we read from frontier', and hence we must ensure that it has the CAS flag enabled. If the read is successful, then the write is placed as the immediate total coherence order successor of the read source.

Instantiating PSO, TSO, SC Given store buffers and main memory, we consider an example to illustrate how past, present, and future are to be interpreted in the context of TSO and PSO.

Example 4. In the illustration, we mark messages which are currently in the main memory with a green ring. These have CAS flag enabled. We start from a state where p_1 only has writes to x in the buffer, and p_2 only has writes to y in its buffer. The main memory writes to x and y are from p_2 and p_1 respectively. If a process has a write to a variable in its buffer, the main memory write to that variable is redundant to it.
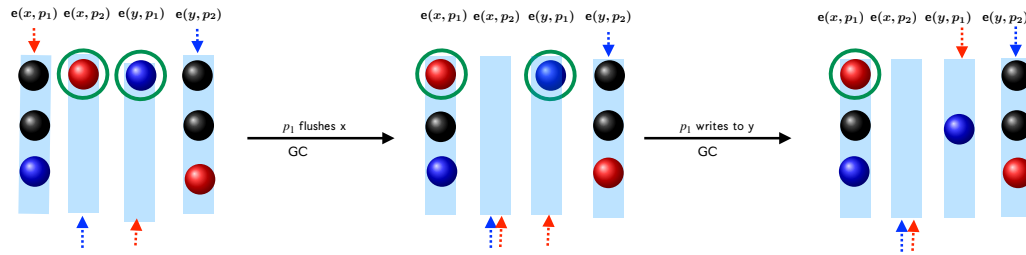
On the other hand, a write in the buffer is in the future of all other processes. Now, p_1 flushes its write to x to the main memory map. This makes the message in $e(x, p_2)$ redundant. p_2 will now source its reads on x from the write that has just been flushed, since its own buffer is empty. Now, we have p_1 making a write to y. This write is placed in the buffer, and hence, remains in the *future* of p_2. Consider the old message in $e(y, p_1)$. Although it is in the main memory, it is no longer accessible, since everyone has a write to y in their buffer. The fact that all frontiers have moved past it also reflects this. Hence, it is redundant, and discarded by garbage collection.

Writes We now describe a write v_{new} to x by p. We make the following sequential updates. ▷ We have $\mathsf{frontier}(p, x, p) \leftarrow v_{new}$

▷ If, for some $p' \neq p$, $\mathsf{frontier}(p, x, p')$ is a message, we do $\mathsf{frontier}(p, x, p') \leftarrow \mathsf{frontier}(p, x, p').\mathsf{nextpos}$. This scenario arises when the buffer of p was hitherto empty, and p was sourcing its reads from the main memory. This update ensures that the message to x in the main memory map becomes redundant to p, but other "flushes" to main memory are yet to be done, because they still reside in the local buffers of other processes.

write PSO, TSO

$$\mathsf{v}_{new}.\mathsf{value} = \mathsf{d}$$

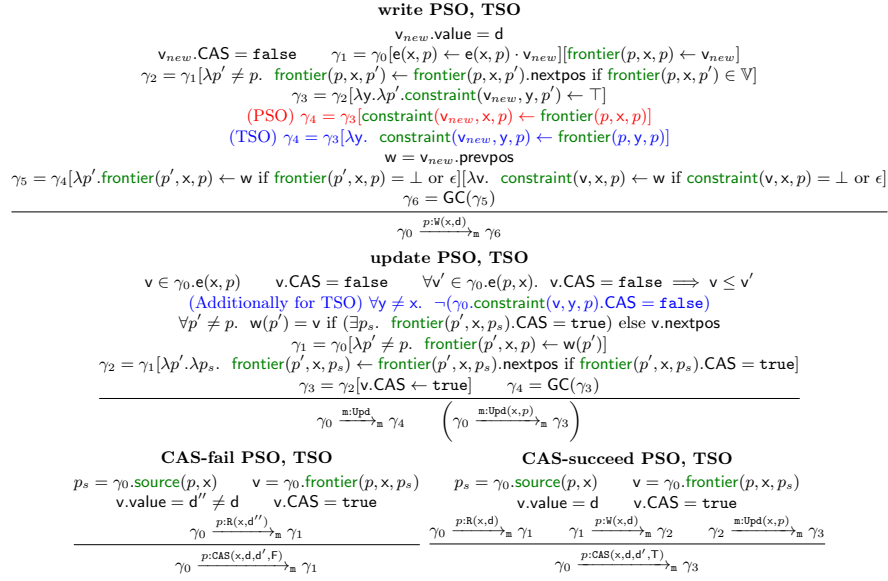$\mathsf{v}_{new}.\mathtt{CAS} = \mathtt{false} \qquad \gamma_1 = \gamma_0[\mathsf{e}(\mathsf{x},p) \leftarrow \mathsf{e}(\mathsf{x},p) \cdot \mathsf{v}_{new}][\mathsf{frontier}(p,\mathsf{x},p) \leftarrow \mathsf{v}_{new}]$

$\gamma_2 = \gamma_1[\lambda p' \neq p. \ \ \mathsf{frontier}(p,\mathsf{x},p') \leftarrow \mathsf{frontier}(p,\mathsf{x},p').\mathsf{nextpos} \text{ if } \mathsf{frontier}(p,\mathsf{x},p') \in \mathbb{V}]$

$\gamma_3 = \gamma_2[\lambda\mathsf{y}.\lambda p'.\mathsf{constraint}(\mathsf{v}_{new},\mathsf{y},p') \leftarrow \top]$

$\text{(PSO) } \gamma_4 = \gamma_3[\mathsf{constraint}(\mathsf{v}_{new},\mathsf{x},p) \leftarrow \mathsf{frontier}(p,\mathsf{x},p)]$

$\text{(TSO) } \gamma_4 = \gamma_3[\lambda\mathsf{y}. \ \ \mathsf{constraint}(\mathsf{v}_{new},\mathsf{y},p) \leftarrow \mathsf{frontier}(p,\mathsf{y},p)]$

$$\mathsf{w} = \mathsf{v}_{new}.\mathsf{prevpos}$$

$\gamma_5 = \gamma_4[\lambda p'.\mathsf{frontier}(p',\mathsf{x},p) \leftarrow \mathsf{w} \text{ if } \mathsf{frontier}(p',\mathsf{x},p) = \bot \text{ or } \epsilon][\lambda\mathsf{v}. \ \ \mathsf{constraint}(\mathsf{v},\mathsf{x},p) \leftarrow \mathsf{w} \text{ if } \mathsf{constraint}(\mathsf{v},\mathsf{x},p) = \bot \text{ or } \epsilon]$

$$\gamma_6 = \mathsf{GC}(\gamma_5)$$

$$\gamma_0 \xrightarrow{p:\mathtt{W}(\mathtt{x},\mathtt{d})}_\mathtt{m} \gamma_6$$

update PSO, TSO

$\mathsf{v} \in \gamma_0.\mathsf{e}(\mathsf{x},p) \qquad \mathsf{v}.\mathtt{CAS} = \mathtt{false} \qquad \forall \mathsf{v}' \in \gamma_0.\mathsf{e}(p,\mathsf{x}). \ \ \mathsf{v}.\mathtt{CAS} = \mathtt{false} \implies \mathsf{v} \leq \mathsf{v}'$

$\text{(Additionally for TSO) } \forall \mathsf{y} \neq \mathsf{x}. \ \ \neg(\gamma_0.\mathsf{constraint}(\mathsf{v},\mathsf{y},p).\mathtt{CAS} = \mathtt{false})$

$\forall p' \neq p. \ \ \mathsf{w}(p') = \mathsf{v} \text{ if } (\exists p_s. \ \ \mathsf{frontier}(p',\mathsf{x},p_s).\mathtt{CAS} = \mathtt{true}) \text{ else } \mathsf{v}.\mathsf{nextpos}$

$\gamma_1 = \gamma_0[\lambda p' \neq p. \ \ \mathsf{frontier}(p',\mathsf{x},p) \leftarrow \mathsf{w}(p')]$

$\gamma_2 = \gamma_1[\lambda p'.\lambda p_s. \ \ \mathsf{frontier}(p',\mathsf{x},p_s) \leftarrow \mathsf{frontier}(p',\mathsf{x},p_s).\mathsf{nextpos} \text{ if } \mathsf{frontier}(p',\mathsf{x},p_s).\mathtt{CAS} = \mathtt{true}]$

$\gamma_3 = \gamma_2[\mathsf{v}.\mathtt{CAS} \leftarrow \mathtt{true} \qquad \gamma_4 = \mathsf{GC}(\gamma_3)]$

$$\gamma_0 \xrightarrow{\mathtt{m}:\mathtt{Upd}}_\mathtt{m} \gamma_4 \qquad \left(\gamma_0 \xrightarrow{\mathtt{m}:\mathtt{Upd}(\mathtt{x},p)}_\mathtt{m} \gamma_3\right)$$

CAS-fail PSO, TSO

$p_s = \gamma_0.\mathsf{source}(p,\mathsf{x}) \qquad \mathsf{v} = \gamma_0.\mathsf{frontier}(p,\mathsf{x},p_s)$

$\mathsf{v}.\mathsf{value} = \mathsf{d}'' \neq \mathsf{d} \qquad \mathsf{v}.\mathtt{CAS} = \mathtt{true}$

$$\gamma_0 \xrightarrow{p:\mathtt{R}(\mathtt{x},\mathtt{d}'')}_\mathtt{m} \gamma_1$$

$$\gamma_0 \xrightarrow{p:\mathtt{CAS}(\mathtt{x},\mathtt{d},\mathtt{d}',\mathtt{F})}_\mathtt{m} \gamma_1$$

CAS-succeed PSO, TSO

$p_s = \gamma_0.\mathsf{source}(p,\mathsf{x}) \qquad \mathsf{v} = \gamma_0.\mathsf{frontier}(p,\mathsf{x},p_s)$

$\mathsf{v}.\mathsf{value} = \mathsf{d} \qquad \mathsf{v}.\mathtt{CAS} = \mathtt{true}$

$$\gamma_0 \xrightarrow{p:\mathtt{R}(\mathtt{x},\mathtt{d})}_\mathtt{m} \gamma_1 \qquad \gamma_1 \xrightarrow{p:\mathtt{W}(\mathtt{x},\mathtt{d})}_\mathtt{m} \gamma_2 \qquad \gamma_2 \xrightarrow{\mathtt{m}:\mathtt{Upd}(\mathtt{x},p)}_\mathtt{m} \gamma_3$$

$$\gamma_0 \xrightarrow{p:\mathtt{CAS}(\mathtt{x},\mathtt{d},\mathtt{d}',\mathtt{T})}_\mathtt{m} \gamma_3$$

Fig. 16: Transitions for PSO, TSO

▷ The constraint, however, differs: In PSO, only $\mathsf{constraint}(\mathsf{v}_{new},\mathsf{x},p) \leftarrow v_{new}$. For all other (y,p'), $\mathsf{constraint}(\mathsf{v}_{new},\mathsf{y},p') \leftarrow \top$. The constraint is rather weak, and the strength of the model is indeed captured through the store buffer semantics and update step. In PSO, writes on different variables by the same process may race. Hence, the update flushes all channels independently. In TSO, for all y, $\mathsf{constraint}(\mathsf{v}_{new},\mathsf{y},p) \leftarrow \mathsf{frontier}(p,\mathsf{y},p)$. For all other p', and all y, $\mathsf{constraint}(\mathsf{v}_{new},\mathsf{y},p') \leftarrow \top$. TSO differs from PSO by requiring a total order on writes issued by a process. We record this total (program) order using constraint. This conveys that a message may be flushed from the buffer only if all the messages that were written by the process before it have also been flushed.

▷ For all other threads p', if $\mathsf{frontier}(p',\mathsf{x},p) = \epsilon$ or $\bot$, then we update $\mathsf{frontier}(p',\mathsf{x},p) \leftarrow \top$ if ϵ and $\mathsf{v}_{new}.\mathsf{prevpos}$ if $\bot$ i.e. if the channel was empty, or the frontier considered all existing messages in the past, then we update the frontier to consider our new message as the start of the future.

▷ Similarly, we correct the other type of reference and ensure it doesn't remain overly strong in view of the new message: for all other v, if $\mathsf{constraint}(\mathsf{v},\mathsf{x},p) = \epsilon$ (it can never be $\bot$): $\mathsf{constraint}(\mathsf{v},\mathsf{x},p) \leftarrow \top$. Finally, we run garbage collection.

Updates In PSO and TSO, the update step "flushes" the store buffer. In the Store-Buffer semantics [45], the first message in the buffer is propagated to the memory. In PSO writes on different variables may overtake each other, while this

is not possible in TSO. In our case, constraint tracks the order of writes made by a process. Messages in the main memory are the only ones with the CAS flag enabled. The update step makes changes to the configuration only if there are messages in the local buffers, i.e. with a `false` CAS flag.

▷ We pick the first message v in $\mathsf{e}(\mathsf{x}, p)$ with CAS flag false, as the message to flush. TSO additionally requires that this be the first buffer message by the process. We check this by further requiring that the constraint for this message on all $\mathsf{e}(\mathsf{y}, p)$ with $\mathsf{y} \neq \mathsf{x}$ is not a buffer message.

▷ All other processes that had frontiers at the main memory message of x (only possible if their buffer had no x messages), will now have a frontier at v. This reflects that v is going to be the new main memory message. If other processes have x writes in their buffers, then their frontier is going to be just after v.

▷ Now, we move all frontiers past the old main memory message to x, making it redundant. Then the CAS flag of v is set to true followed by garbage collection.

CAS In order for p to execute a CAS on variable x, PSO requires p to have no buffer messages on x. TSO further requires p to have no buffer messages at all. A CAS is equivalent to reading, and writing iff the read value matches the expected value. The write is immediately flushed to memory.

Finally, **SC is stronger than TSO**: in SC, every write is immediately flushed to the memory.

B Decisive Markov Chain Algorithms

B.1 Computing Quantitative Control State Reachability using Control State Reachability

We fix a Markov chain $\mathcal{C}$ that refines the underlying transition induced by a program. Additionally we assume that the Markov chain satisfies the probabilistic fairness property, i.e. plain configurations are visited infinitely often with probability one. In Markov chain parlance [6], we say that the set of plain configurations is a *finite attractor*. An important property of Markov chains with finite attractors is that they are *decisive* with respect to every set of configurations G. We summarise the two crucial properties from which the correctness and termination of our algorithms immediately follow.

- Definition of decisiveness: for any set G, an infinite run almost certainly eventually either reaches G, or a configuration γ from which G is unreachable.
- (Strong) Markov chain fairness: if γ_1 is reachable from γ_0, then any run that visits γ_0 infinitely often almost certainly visits γ_1 (infinitely often).

We now present an algorithm that allows us to leverage the known decidability results for non-probabilistic versions of reachability (Problems 2). Such techniques have been used in the context of operational TSO [5]. The algorithm approximates the probability of repeatedly reaching the target configuration by maintaining an over and an under-approximation of this probability, which are initialised to 0 and 1 respectively. At a high level the algorithm performs *forward reachability* in a breadth-first manner, which is implemented using a FIFO queue (Queue).

The queue maintains pairs consisting of a configuration and a probability of a path leading up to that configuration. If the current configuration γ is such that every plain configuration reachable from it can in turn reach the target control state c_t, then we know that the run from this point is guaranteed to visit c_t infinitely often. This is due to the attractiveness of plain configurations, and Markov chain fairness. On the other hand, on reaching a state from which c_t is not reachable, we can conclude this is a path that visits c_t only finitely often. The algorithm terminates when the difference in the upper and lower estimates is less than ε. This implies that both the estimates are ε close to the exact value, and that we can return the lower estimate while maintaining correctness (line 14). Termination of this algorithm relies on (a) the fact that the limiting value of the difference of the two estimates (as the number of iterations goes to infinity) is zero and (b) that the check performed on line 8 terminates.

(a) is a result of the fact that the Markov chain for which we consider this problem is decisive. The set G is the set of configurations γ is such that every plain configuration reachable from it can in turn reach the target control state c_t. If G becomes inaccessible, it is always detected by eventually reaching a plain configuration from which c_t isn't reachable. (b) rests on the decidability of the corresponding reachability problem. This is precisely the reachability prob-

---
Algorithm 1: Quantitative Control State Reachability

Data: prog: concurrent program, c_t: target control state, γ_{init}: initial
   configuration, ε: accuracy
Result: Probability of reaching l infinitely often, precise to within ε

1   underAppr $\leftarrow 0$
2   overAppr $\leftarrow 1$
3   Queue $\leftarrow [(\gamma_{\text{init}}, 1)]$
4   **while** overAppr $-$ underAppr $> \varepsilon$ **do**
5 $(\gamma, \phi) \leftarrow$ Queue.pop()
6   guaranteeFlag $\leftarrow$ true
7   **foreach** $\gamma' \in \Gamma_{plain}$ **do**
8   **if** $(\gamma \xrightarrow{*} \gamma') \wedge \neg(\gamma' \xrightarrow{*} c_t)$ **then**
9   guaranteeFlag $\leftarrow$ false
10   **if** guaranteeFlag **then**   underAppr $\leftarrow$ underAppr $+ \phi$
11   **else if** $\neg(\gamma \xrightarrow{*} c_t)$ **then**   overAppr $\leftarrow$ overAppr $- \phi$
12   **else**
13   **foreach** γ' *such that* $\gamma \rightarrow \gamma'$ *with probability* $p > 0$ **do**
     Queue.push$(\gamma', \phi \cdot p)$
14 **return** underAppr
---

lem from arbitrary configurations. Consequently, we get the following general result.

Theorem 4. *If Problem 2 is decidable for a memory model, then Problem 3 is decidable for Markov chains that satsify probabilistic memory fairness.*