Recurrence Sets for Proving Fair Non-termination under Axiomatic Memory Consistency Models

THOMAS HAAS, TU Braunschweig, Germany
ROLAND MEYER, TU Braunschweig, Germany
HERNÁN PONCE DE LEÓN, Huawei Dresden Research Center, Germany
ANDRÉS LOMELÍ GARDUÑO, Huawei Dresden Research Center, Germany

Recurrence sets characterize non-termination in sequential programs. We present a generalization of recurrence sets to concurrent programs that run on weak memory models. Sequential programs have operational semantics in terms of states and transitions, and classical recurrence sets are defined as sets of states that are existentially closed under transitions. Concurrent programs have axiomatic semantics in terms of executions, and our new recurrence sets are defined as sets of executions that are existentially closed under extensions.

The semantics of concurrent programs is not only affected by the memory model, but also by fairness assumptions about its environment, be it the scheduler or the memory subsystems. Our new recurrence sets are formulated relative to such fairness assumptions. We show that our recurrence sets are sound for proving fair non-termination on all practical memory models, and even complete on many.

To turn our theory into practice, we develop a new automated technique for proving fair non-termination in concurrent programs on weak memory models. At the heart of this technique is a finite representation of recurrence sets in terms of execution-based lassos. We implemented a lasso-finding algorithm in Dartagnan, and evaluated it on a number of programs running under CPU and GPU memory models.

CCS Concepts: • **Theory of computation → Concurrency**; **Program verification**; *Verification by model checking*.

Additional Key Words and Phrases: Non-termination, axiomatic memory models, forward progress, automatic verification

ACM Reference Format:
Thomas Haas, Roland Meyer, Hernán Ponce de León, and Andrés Lomelí Garduño. 2026. Recurrence Sets for Proving Fair Non-termination under Axiomatic Memory Consistency Models. *Proc. ACM Program. Lang.* 10, POPL, Article 45 (January 2026), 35 pages. https://doi.org/10.1145/3776687

1 Introduction

Non-termination in concurrent programs is a significant challenge, arguably even more so than in sequential programs. This is because thread synchronization often relies on waiting mechanisms, which are typically implemented using unbounded loops and can potentially lead to non-termination. What makes this problem particularly hard is that the memory subsystem over which the threads communicate may exhibit weak behaviors. It may reorder load and store requests, buffer them in thread-local storage, and even asynchronously propagate stores (i.e., one thread may see a new value in memory, while another still sees the old one). All of these behaviors impact the program's semantics and, if not accounted for, may lead to non-termination bugs [18, 34].

Authors' Contact Information: Thomas Haas, TU Braunschweig, Braunschweig, Germany, t.haas@tu-bs.de; Roland Meyer, TU Braunschweig, Braunschweig, Germany, roland.meyer@tu-bs.de; Hernán Ponce de León, Huawei Dresden Research Center, Dresden, Germany, hernanl.leon@huawei.com; Andrés Lomelí Garduño, Huawei Dresden Research Center, Dresden, Germany, andreslomeli02@gmail.com.

ACM 2475-1421/2026/1-ART45
https://doi.org/10.1145/3776687

Different CPU and GPU architectures, and even programming languages like C11 and Java, come with different guarantees about how their memory subsystem behaves, commonly described by their so-called memory consistency models [5, 8, 29, 31, 40].

To further complicate the matter, the termination of a concurrent program also depends on fairness assumptions [14]. A memory subsystem that never propagates stores or that infinitely reorders/delays requests past other ones would be considered unfair. However, in most real systems, the memory subsystem can be assumed to be fair. In a similar vein, fairness assumptions about the scheduler are also crucial: if a thread that everyone is waiting for never gets scheduled, the program will hang. Unlike memory fairness, there do exist real systems which exhibit unfair scheduling behavior and programs need to be tolerant about it. For example, GPUs generally provide only weak scheduling (also known as forward progress) guarantees. The difference in forward progress guarantees is known to cause some parallel algorithms to fail to be portable across different GPU architectures [24, 43, 44]. Progress guarantees also exist on the language level, for example, C++ does not require that a language implementation provides fair progress for threads, not even the main thread (however, it encourages implementations to do so).

Non-termination of single-threaded systems is characterized by the existence of a certain proof object, a so-called recurrence set, first introduced by Gupta et al. [15]. Intuitively, a recurrence set is a set of states that is reachable and each of its states has a transition that stays inside. Naturally, we would like to have a generalization of recurrence sets to concurrent systems that takes into account all of the aforementioned issues, that is, the memory consistency model and the fairness assumptions about the memory and the scheduler. This is precisely the contribution of our work.

The main difficulty in coming up with this generalization is the handling of the memory consistency model. The reason is that memory models are commonly formulated in an axiomatic style, that reasons about whole program executions rather than single states and transitions [3, 6]. Indeed, a key point of axiomatic semantics of concurrent programs is the lack of a classical notion of shared state. Instead, the semantics views a program execution as a graph of interactions and communications between threads.

To overcome this problem, our first contribution is to move from state-based to execution-based recurrence sets. Intuitively, such a recurrence set will consist of ever larger prefixes of executions that approximate an infinite, and thus non-terminating, execution. These execution-based recurrence sets will give us a sound, and in many cases even complete, characterization of non-termination relative to the memory model and fairness assumptions of interest.

While the above generalization is interesting by itself, state-based recurrence sets have algorithmic advantages. State-based recurrence sets can be finite, in which case they represent a non-terminating run where some finite set of states is repeatedly visited (also known as a lasso). Even if they are not finite, they can often be represented by finite means using symbolic representations. Both cases enable the usage of automatic techniques to prove non-termination [11, 15, 33]. In contrast, our execution-based recurrence sets cannot be finite since they must contain ever-larger execution prefixes. Yet, to our knowledge, there do not exist readily applicable techniques that could yield finite representations for infinite sets of such executions.

Our second contribution tackles this problem by developing a suitable abstraction. Our key idea is to move from execution prefixes to execution infixes, i.e., snippets of executions, by abstracting parts of the execution that are "sufficiently far in the past". This will allow us to obtain execution-based lassos that represent a finite set of repeating infixes, rather than a finite set of repeating states. In the extreme case, where we abstract down to minimally-sized infixes, we will recover classical state-based recurrence sets and lassos. This is no coincidence: a state can be understood as a particularly small infix of a program execution. Therefore, our execution-infix-based recurrence sets are a proper generalization of the classical ones.

```
T1:
x = -1;
while(y != 1) {
  x = 0;
  x = 1;
}

T2:
while(x != 0);
y = 1;
```

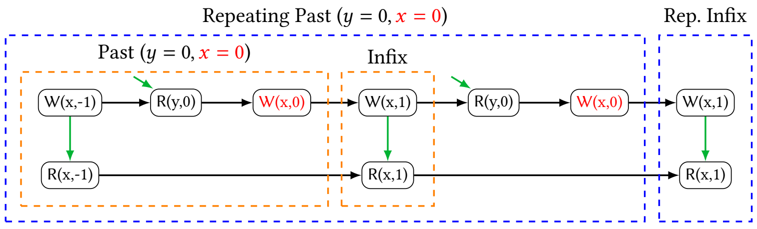

Fig. 1. A non-terminating program under weak fairness (left) and an execution-based lasso (right). Two infixes repeat with a common past (modulo abstraction).

We illustrate execution-based lassos in Figure 1. T1 initially writes $x = -1$ and then starts a loop that writes $x = 0$ followed by $x = 1$ in each iteration, until it gets the signal to terminate ($y = 1$) from T2. T2 waits until it observes $x = 0$ from T1, and once it does, it writes $y = 1$, signaling T1 to stop, and then terminates. Under weak fairness assumptions, T2 may always observe the second write $x = 1$ of each iteration of T1 and skip over $x = 0$. The execution-based lasso that captures this non-terminating behavior is given on the right of the figure. The fact that T2 repeatedly observes $x = 1$ from T1 is captured by the repeating infix of the lasso. The past that lead to the first repetition of the infix is equivalent to the past that lead to the second repetition because, roughly speaking, both pasts produce the same final memory state (highlighted in red).

Our final contribution is an extension to DARTAGNAN [36–38], a bounded model checker for weak memory concurrency, that automatically finds execution-based lassos in programs using SMT-based bounded model checking. This enables us to automatically prove the lack of portability (due to non-termination) of parallel algorithms in the absence of forward progress guarantees. While the lack of portability is regarded as folklore in the GPU community, to the best of our knowledge, no tool has yet been able to prove it. We also used DARTAGNAN to reproduce known results about the need for barriers in several spinlock implementations to avoid non-termination [39] and how different forward progress guarantees affect the termination of litmus tests [46].

2 Concurrent Programs & Axiomatic Semantics

We present the basic notions of concurrent programs, memory consistency models, and the axiomatic semantics they induce.

2.1 Concurrent Programs

We consider concurrent programs p that are a top-level parallel composition of (possibly infinitely many) sequential programs, the so-called threads. Each thread is identified by a unique thread identifier $t \in \mathbf{Tid}$ and its instructions work over a set of local registers $\mathbf{Reg}_t$ and a set of shared locations $\mathbf{Loc}$. The (local) configuration of a thread is standard: it consists of a program counter $pc \in \mathbf{Pc}_t$ (initially pointing to the first instruction) and a valuation $\lambda \in \mathbf{Reg}_t \to \mathbb{Z}$ of all its registers which are initially zero. We assume that pc may take a special value $\perp$ that indicates termination. Formally, the configurations of a thread are given by the set $\mathbf{Conf}_t := \mathbf{Pc}_t \times (\mathbf{Reg}_t \to \mathbb{Z})$ and we denote its elements by $c = (pc, \lambda)$.

A thread t induces a labeled transition system LTS(t) over configurations. The execution of an instruction emits a (possibly empty) sequence of events that form the labels of the transitions. Events always contain the thread id of the executing thread and the pc of the executed instruction.

The local instructions, such as register assignments and branching, do not emit any events[1] and transform the local state as expected. The memory instructions work as follows.

(i) A store instruction emits a write event $\mathsf{W}(l, v)$ containing the accessed location l and the written value v. It does not change the local state other than incrementing the pc.
(ii) A load instruction $r := \mathsf{load}(l)$ assign a non-deterministically chosen value v to register r, and emits the read event $\mathsf{R}(l, v)$.
(iii) A fence instructions emits a fence event containing the name/kind of the fence.
(iv) Read-modify-write (rmw) instructions come in multiple flavors depending on what local modifications they do and if they are guaranteed to succeed (e.g., a compare-and-swap may fail). In all cases, a rmw can be understood as a load, followed by a sequence of local instructions, and possibly a final store. This instruction sequence gives both the local effect on the state and the emitted event sequence.

A run of thread t is a (possibly infinite) sequence $\rho_t = c_0.x_{0,0} \ldots x_{0,k_0}.c_1.x_{1,0} \ldots x_{1,k_1}.c_2 \ldots$ alternating between configurations and event sequences induced by the LTS. A run is called maximal if it is infinite or ends in a configuration without transitions, i.e., where $pc = \bot$ holds. A run is called initialized if it starts in the initial configuration. We denote the set of runs of a thread by Runs_t.

For a concurrent program $p = \prod_i t_i$ we define its set of runs by $\mathsf{Runs} := \prod_t \mathsf{Runs}_t$ and its set of configurations by $\mathbf{Conf} = \prod_t \mathbf{Conf}_t$. A program run $\rho \in \mathsf{Runs}$, thus, maps each thread $t \in \mathbf{Tid}$ to a run $\rho_t \in \mathsf{Runs}_t$ of that thread. We say that the program run is finite if the total size of all its thread runs is finite; otherwise, it is infinite. We say that the run is initialized if all its thread runs are initialized and it is maximal if it is infinite or all its thread runs are terminating. If the number of threads is finite, then every maximal program run contains a maximal thread run.

If a run ρ_t is the prefix of another run ρ'_t of the same thread t, then we denote this by $\rho_t \sqsubseteq \rho'_t$ and also call ρ' an extension of ρ. Naturally, a program run ρ is a prefix of ρ', also denoted by $\rho \sqsubseteq \rho'$, if we have for each thread t that $\rho(t)$ is a prefix of $\rho'(t)$.

2.2 Anarchic Semantics

Every program run $\rho \in \mathsf{Runs}$ induces an edge-labeled event graph $EG(\rho) = (X, \mathsf{po}, \mathsf{rmw}, \ldots)$. The nodes X of this graph are the events $\mathsf{Ev}(\rho)$ appearing in the run plus a (possibly infinite) set of special initializing write events that write 0 to each memory location. The labeled edges are given in the form of several binary relations over the events X, the so-called base relations, and capture some information of the run in the graph. One such relation is po which relates an event x to y if both are in the same thread and y appears later in the thread's run. Another relation is rmw which relates a read event to a write event if they were emitted from the same transition of an rmw instruction. There exist other base relations that we will introduce when necessary. The set of base relation names is denoted by $\mathbb{R}_B$, and we understand an event graph as giving an interpretation to these base relation names over a common domain X. By abuse of notation, we often use a relation name, say po, to mean either the relation name or the concrete interpretation of that relation name in an event graph if it is clear from context. We write $EG.X$ to denote the events of a graph and if the context is unclear, we write $EG.\mathsf{r}$, $EG(\mathsf{r})$ or $\mathsf{r}(EG)$ to mean the interpretation of r in EG.

An event graph EG is *justifiable* if there exist a read-from relation rf and a coherence relation co satisfying the following properties. Read-from relates write events to read events such that *(i)* each read $r \in EG.X$ has exactly one write that relates to it and *(ii)* whenever $\mathsf{rf}(w, r)$ holds then the values and locations of the related events match. Coherence totally orders all write events to the same location (with init writes always being first) and never relates write events to different locations, i.e., coherence can be understood as a union of total orders, one total order per memory location. A

[1]This simplifies the presentation.

justifiable event graph together with a justification forms an execution graph $XG = (EG, \mathsf{rf}, \mathsf{co})$, which we often write in flattened form $XG = (X, \ldots, \mathsf{rf}, \mathsf{co})$. In many contexts, we understand rf and co as being part of the base relations $\mathbb{R}_B$. A program run ρ is called justifiable if its event graph $EG(\rho)$ is justifiable. An (anarchic) execution of a program p is a pair $\varepsilon = (\rho, XG)$ of an initialized and maximal run together with an execution graph that justifies its corresponding event graph. The anarchic semantics of a program p, denoted by $[\![p]\!]$, is now given by its set of executions.

2.3 Memory Consistency Models

A memory (consistency) model mm is a predicate on execution graphs that tells us if the execution graph XG is considered *consistent*, denoted by denoted by $XG \models mm$, from the perspective of the architecture being modeled, meaning it can occur as behavior. The semantics of a program under a memory model is then defined by $[\![p]\!]_{mm} = \{\varepsilon \in [\![p]\!] \mid \varepsilon.XG \models mm\}$, the subset of program executions that are consistent with the memory model.

We consider memory models formulated in the CAT language [4]. Such a memory model makes judgments purely based on the base relations, e.g. the rf-edges and co-edges in the execution graph, but not on, say, concrete values and addresses communicated by events.

Intuitively, this means that the memory model reasons about the shape of the execution graph. To do so, the CAT language defines so-called derived relations (auxiliary relations), the names of which we denote by $\mathbb{R}_D$, using relation algebra over base relations and other derived relations. We assume that $\mathbb{R}_B$ and $\mathbb{R}_D$ are disjoint and denote their union by $\mathbb{R} := \mathbb{R}_B \cup \mathbb{R}_D$. For example, $\mathsf{fr} := \mathsf{rf}^{-1}; \mathsf{co}$ and $\mathsf{hb} := \mathsf{po} \cup \mathsf{co} \cup \mathsf{rf} \cup \mathsf{fr}$ are derived relations. Over these derived relations, the memory model then imposes constraints such as acyclic(hb), stating that execution graphs are consistent only if the derived hb-relation is acyclic on those graphs. In addition to binary base and derived relations, there exist also unary relations (sets) that denote, for example, the set of write events W or the set of read events R. The complete grammar of the CAT language is given in Figure 2. We use the standard shorthand notations for the reflexive, transitive closure $\mathsf{r}^* := \mathsf{r}^+ \cup \mathsf{id}$ and the negation $\neg\mathsf{r} := (X \times X) \setminus \mathsf{r}$ (or $\neg\mathsf{r} := X \setminus \mathsf{r}$, if r is a set). Notice that CAT permits recursive definitions.

$$
\begin{array}{rcl}
mm & ::= & axm \mid def \mid mm \wedge mm \\
axm & ::= & \mathbf{acyclic}(r) \mid \mathbf{irreflexive}(r) \\
 & & \mid \mathbf{empty}(r) \mid \mathbf{empty}(s) \\
def & ::= & \texttt{let}\ rn := r \mid \texttt{let}\ sn := s \\
r & ::= & br \mid rn \mid r \cup r \mid r \cap r \mid r \setminus r \\
 & & \mid r;r \mid r^{-1} \mid r^{+} \mid [s] \mid s \times s \\
s & ::= & bs \mid sn \mid \mathtt{domain}(r) \mid \mathtt{range}(r) \\
 & & \mid s \cap s \mid s \cup s \mid s \setminus s \\
br & ::= & \mathsf{id} \mid \mathsf{rf} \mid \mathsf{co} \mid \mathsf{po} \mid \ldots \\
bs & ::= & \mathsf{X} \mid \mathsf{W} \mid \mathsf{R} \mid \mathsf{F} \mid \ldots
\end{array}
$$

Fig. 2. CAT Grammar.

Formally, a CAT memory model mm describes a mapping of an interpretation of the base relations $I_B : \mathbb{R}_B \longrightarrow \mathbb{P}(X) + \mathbb{P}(X \times X)$ (over a common domain X) to an interpretation of the derived relations $I_D : \mathbb{R}_D \longrightarrow \mathbb{P}(X) + \mathbb{P}(X \times X)$ and a consistency judgment $J : \mathbb{B} = \{0, 1\}$ where 0 denotes consistency and 1 denotes inconsistency.

Intuitively, the computation of I_D is done by repeatedly evaluating the definitions of the CAT model until a fixed point is reached, and on this fixed point the final consistency judgment is made. Actually, in the presence of difference operators, the order of evaluation matters and should follow a so-called stratification. We leave out the details as they are unimportant for our development.

The meanings of 0 and 1 for the consistency judgment seem reversed. The reason we do this is for technical simplicity: the axioms in CAT are antitonic with respect to the standard ordering (larger relations are more likely to violate an axiom) but we want to see them as monotonic. Alternatively, one could also choose to reverse the ordering on $\mathbb{B}$ and have $1 < 0$.

We can naturally lift mm to a function on executions $mm((\rho, XG)) = mm(XG)$ where XG provides the domain and the interpretations of the base relations over that domain. It is also convenient to think of mm as enriching a given execution (graph) with new relations $\mathsf{r} \in \mathbb{R}_D$. This allows us to write $XG.\mathsf{r}$ (or similar) also for derived relations $\mathsf{r} \in \mathbb{R}_D$ to refer to the interpretation of r obtained by evaluating mm over XG.

To simplify the presentation in the remaining paper, we will assume that there are only binary relations. After all, we can identify a set by the (binary) identity relation on that set, thereby lifting all sets to binary relations.

3 Fairness

The concurrent programs we have defined so far may exhibit unfair behavior where certain actions they could take are delayed indefinitely. Such unfair behavior comes in many flavors: an active thread is never scheduled, a buffered store is never flushed, a reorderable instruction is reordered past infinitely many others, an instruction that may spuriously fail never succeeds, and so on. A typical (idealized) assumption is that real concurrent programs do not exhibit such unfair behavior and so they should not be considered when the system is analyzed w.r.t. non-termination. That being said, sometimes (partially) unfair behavior is expected. For example, concurrent programs executed on GPUs may experience unfair scheduling [45]. This is in contrast to programs executed on CPUs where the OS-provided scheduler is expected to guarantee fairness. Seeing that fairness is crucial to non-termination of concurrent programs, we need to model it.

3.1 Scheduler Fairness

We define scheduler fairness by a function that tells us, at any given moment in time, which threads are subject to fair scheduling. Unlike traditional operational schedulers, the scheduler fairness function does not tell which thread performs the next transition, but rather which threads are guaranteed to eventually make a step if their fairness status is not revoked. We make this formal.

For a finite program run ρ we define its final configuration $c(\rho) \in \mathbf{Conf}$ to be the program configuration obtained by taking the final thread configuration $c(\rho(t))$ of each thread run. A scheduler function $\sigma : \mathbf{Conf} \to \mathbb{P}(\mathbf{Tid})$ maps configurations to a (possibly empty) set of threads, the threads which are currently subject to fair scheduling. We extend the definition of σ to finite runs by $\sigma(\rho) := \sigma(c(\rho))$. We further extend it to infinite runs ρ via $\sigma(\rho) := \bigcap_{\rho_i \to \rho} \liminf_i \sigma(\rho_i) = \bigcap_{\rho_i \to \rho} \bigcup_i \bigcap_{j \geq i} \sigma(\rho_j)$, where $\bigcap_{\rho_i \to \rho}$ denotes an intersection over all increasing sequences $(\rho_i)_{i \in \mathbb{N}}$ of finite runs that converge to ρ. We explain this definition. A thread $t \in \sigma(\rho)$ is subject to fair scheduling if in all sequences of finite approximations of ρ, there is a point from which onward t is continuously subject to fair scheduling. Now we say that an infinite run ρ is σ-fair if for all threads $t \in \mathbf{Tid}$ we have $t \in \sigma(\rho)$ implies that $\rho(t)$ is a maximal run in LTS(t). Naturally, we can lift the scheduler function to executions via $\sigma(\varepsilon) := \sigma(\varepsilon.\rho)$ and talk about σ-fair executions.

We give a few examples of scheduler functions σ. Fully unfair scheduling is given by $\sigma(c) = \emptyset$ and fully fair scheduling is given by $\sigma(c) = \mathbf{Tid}$. For GPUs, other scheduler models have been proposed in the literature [45, 46]. One of the proposed models is called OBE, where every thread that has taken at least one step experiences fair scheduling, that is, $\sigma_{OBE}(c) = \{t \mid c(t).pc \neq pc_{init}\}$. Another model for GPUs is HSA, where the active thread with the lowest id is subject to fairness, i.e., $\sigma_{HSA}(c) = \{\min_t\{t \mid c(t).pc \neq \bot\}\}$. Similar to OBE, linear occupancy-bound execution (LOBE) guarantees fair scheduling to any thread that has taken a step, or if its id is smaller than a thread that has taken a step, i.e., $\sigma_{LOBE}(c) = \{t \mid \exists t' \geq t : c(t').pc \neq pc_{init}\}$. A straightforward approach to create new fairness properties from existing ones is to consider the disjunction of their guarantees. This idea has been used to define the HSA+OBE model [45].

3.2 Memory Fairness on Execution Graphs

Memory fairness captures the idea that actions/events are not delayed indefinitely. Intuitively, an infinitely delayed event must have infinitely many events happening before it. In terms of execution graphs, this means that the event has infinitely many predecessors in, say, a happens-before relation hb. Therefore, memory fairness can be characterized by the absence of events with infinitely many predecessors. We make this formal.

Let XG be an infinite execution graph and let r be a relation on $XG.X$. We define the r-prefix of an event y by $\textsc{prefix}(\mathsf{r}, y) := \{x \mid (x, y) \in \mathsf{r}^+\}$. We say that XG is r-(memory-)fair, if r is prefix-finite in XG, meaning that for all $y \in XG.X$ we have that $\textsc{prefix}(\mathsf{r}, y)$ is finite. It is R-(memory-)fair if it is r-fair for every $\mathsf{r} \in R$.

Lahav et al.[22] characterized memory fairness for several memory models as $\{\mathsf{co}, \mathsf{fr}\}$-fairness. Intuitively, the former says that no store is buffered indefinitely and the latter says that each store eventually propagates to all threads. Indeed, for some models it may be necessary to also require hb-fairness to prevent infinite reordering. It is not precisely clear what would be the right notion of memory fairness for each memory model; a reason why we keep the theory fully general.

We remark that the above notion of memory fairness is considered weak, which we explain on the introductory example in Figure 1. Without memory fairness, T2 may read $x = -1$ indefinitely, never seeing any of the infinitely many other stores of T1. Under (weak) memory fairness (concretely fr-fairness), T2 cannot read $x = -1$ forever, but it may forever observe the infinitely many instances of $x = 0$, always skipping over $x = 1$, and therefore fail to terminate. Under strong(er) memory fairness assumptions such as presented in [1], T2 must eventually observe $x = 1$ and thus terminate.

3.3 Other Fairness Considerations

Some instructions in a program may fail spuriously, such as a weak CAS in C(++) or a load-linked/store-conditional (LL/SC) pair on hardware. The platform typically guarantees forward progress for those instructions, meaning that if they are performed in a retry-loop, they will eventually succeed and the loop will terminate. This is a type of fairness assumption that we mention for the sake of completeness but otherwise ignore in the remainder of the paper. Nevertheless, including it in our main theory is straightforward.

4 Recurrence Sets

Our goal is to characterize non-terminating behavior in concurrent programs. Our approach is to take the classical notion of recurrence sets - a complete characterization of non-termination - and adapt it to the axiomatic weak memory setting. To this end, we recall what recurrence sets are.

Definition 4.1 (Operational Recurrence Set). A recurrence set is a set of states Rec that is *(i)* reachable from the initial states and *(ii)* every state inside has a transition leading back into it.

It is easy to see that the existence of recurrence sets gives rise to non-terminating runs: by property *(i)* the system can reach the recurrence set and by property *(ii)* it can stay forever inside. Conversely, the states observed along any non-terminating run form a recurrence set. Therefore, recurrence sets are complete.

4.1 Generalizations

Now let us discuss how the adaption to weak memory works and what problems we need to tackle. The first problem we encounter is that axiomatic semantics does not reason about states but rather runs/executions. This conceptual gap, however, is easy to bridge: we can redefine classical recurrence sets to be a set of (finite) linear runs rather than a set of states, and let the transition relation extend the runs by appending new states. Indeed, such run-based recurrence sets are equivalent to state-based recurrence sets in the sense that the existence of one implies the existence of the other. If we now replace the linear runs by partially ordered ones, we get a definition that is more suitable for axiomatic weak memory. Unfortunately, this relaxation is not enough.

A key feature of axiomatic semantics is that we only need to justify maximal runs by constructing an execution graph. However, we cannot build run-based recurrence sets from maximal runs only. To reconcile this discrepancy, we define execution (graph) prefixes that may provide only partial justifications for initialized (non-maximal) runs. This gives rise to execution-based recurrence sets.

The second problem we face is that classical recurrence sets lack the concept of fairness entirely. We incorporate scheduler fairness and memory fairness into our definition of recurrence sets to obtain *fair recurrence sets*.

We shortly sketch the definitions we need in order to formulate fair recurrence sets and state our main theorem. Later we will make the definitions precise. An execution prefix is a (possibly non-maximal) run together with a partial execution graph that may contain so-called unjustified reads, whose corresponding write partner will appear in the future of the run. Execution prefixes can be extended by extending the run and its execution graph accordingly. We want to construct a sequence of increasing execution prefixes that converges to an infinite, consistent, and fair execution. To ensure memory fairness in this construction, we need to make sure that for each memory-fair relation r and each event x, we eventually stop increasing the r-prefix of x when extending the execution prefixes. To do so, we add a prefix-completeness marker π that keeps track of those events whose r-prefix is complete for each memory-fair relation r. We then only consider extensions that do not touch the prefix of already marked events, and call them memory-fair extensions. Lastly, to ensure scheduler fairness, we need to make sure that fairly-scheduled threads get extended. We say that an extension $\rho \sqsubseteq \rho'$ σ-progresses a non-terminated thread t, if $t \in \sigma(\rho') \implies \rho(t) \sqsubsetneq \rho'(t)$, meaning t's run gets extended if its fairness status is not revoked. We naturally lift this to notion to extensions of executions.

Definition 4.2 (Fair recurrence sets). Let p be a program, mm a memory model, σ a scheduler fairness function, and $R \subseteq \mathbb{R}$ a set of relations of mm. A fair recurrence set Rec is a non-empty set of finite execution prefixes of the program p satisfying the following properties for each $\varepsilon \in$ Rec:

(i) ε has a proper and consistent extension $\varepsilon \sqsubseteq \varepsilon' \in$ Rec.
(ii) If ε has an unjustified read, then there is an extension that justifies this read.
(iii) For each non-terminated thread $t \in \sigma(\varepsilon)$ there is an extension that σ-progresses t.
(iv) For each relation $\mathsf{r} \in R$ and event $x \in \varepsilon.X$ there is an extension $\varepsilon \sqsubseteq \varepsilon'$ such that $x \in \varepsilon'.\pi(\mathsf{r})$.
(v) All of the above extensions are memory-fair.

We sketch the soundness of fair recurrence sets, that is, the existence of a fair recurrence set implies the existence of a fairly non-terminating execution. The idea is that we start with any execution prefix $\varepsilon \in$ Rec and iteratively use properties *(i)*-*(iv)* to extend this prefix. In this way, we construct an increasing sequence of execution prefixes that converges to a σ-fair, R-fair, and consistent infinite execution in the limit. The scheduler fairness is guaranteed by property *(iii)* and the memory fairness is guaranteed by properties *(iv)* and *(v)*. To formally prove this, we will need to study the properties of such sequences. As it turns out, the soundness is not unconditional: the memory model of interest is required to be *lower semi-continuous*, a property that ensures that consistency and memory fairness behave nicely in the limit. We give the definition later.

Theorem 4.3 (Fair recurrence sets are sound). *Let Rec be a fair recurrence set for program p and lower semi-continuous memory model mm. Then there is a fair and non-terminating execution of p that is consistent with mm.*

Fortunately, it turns out that many memory models are lower semi-continuous.

Claim 1 (Practical memory models are lower semi-continuous). *The following memory models are lower semi-continuous: SC, TSO, ARM8, Power, RISCV, RC11, IMM, LKMM, NVIDIA PTX, Vulkan, and OpenCL. We believe this holds true for all practical memory models.*

Indeed, most memory models are even monotonic (defined as expected in the next section). For those memory models, fair recurrence sets are complete. Roughly, the argument for operational recurrence sets works again: for a fair, consistent, and non-terminating execution, the set of its prefixes forms a recurrence set.

Theorem 4.4 (Completeness for monotonic memory models). *Let p be a program, mm a monotonic memory model, σ a scheduler function, and R a set of memory-fair relations. If p admits a consistent and fair non-terminating execution, then there exists a fair recurrence set for p.*

We spend the rest of this section making the above sketched definitions precise. In the next section we will then formally prove the soundness theorem Theorem 4.3.

4.2 Definitions

As we have discussed above, we want to define recurrence sets in terms of execution prefixes, that is, non-maximal, initialized runs with partial justifications. The reason why we call them "prefixes" is because of how we imagine their construction. Let us explain this. Consider an infinite execution $\varepsilon = (\rho, XG)$ and a (finite) prefix $\rho' \sqsubseteq \rho$ of its run. Suppose we restrict XG to those events appearing in ρ' and call the resulting graph XG'. We state three facts about the restricted execution graph XG'. First, it is a po-prefix of the original execution graph. Second, one can show that $EG(\rho').\mathsf{r} \subseteq XG'.\mathsf{r}$ for all base relations r besides coherence and read-from. Lastly, the restricted coherence order $XG'.\mathsf{co}$ is still total per memory location and so it satisfies the properties of a justification. However, the restricted read-from relation $XG'.\mathsf{rf}$ may fail to justify each read: a read may have lost its matching write event. In fact, the existence of such unjustified reads cannot be avoided by, say, only considering special prefixes of the infinite execution. Memory models that allow for $(\mathsf{po} \cup \mathsf{rf})$-cycles may admit infinite consistent executions such that the restriction to any finite prefix contains unjustified reads. Therefore, our notion of execution prefix will have to permit such unjustified reads with the idea that they have to eventually get justified.

Definition 4.5 (Partial execution graph). Let ρ be a run. We define a partial execution graph $XG = (EG(\rho), \mathsf{rf}, \mathsf{co})$ for ρ like an execution graph for ρ with the relaxation that not all read events need to have an rf-edge. Read events without an rf-edge are called *unjustified reads*.

We remark that a partial execution graph with no unjustified reads is just a standard execution graph. The notion of consistency of execution graphs naturally extends to partial execution graphs: we simply evaluate mm on the partial execution graph.

We now define the aforementioned prefix-completeness marker that we need to ensure memory fairness in our recurrence sets.

Definition 4.6 (Prefix-completeness marker). Let $R \subseteq \mathbb{R}$ be a set of relation names and XG a partial execution graph. A prefix-completeness marker is a function $\pi : R \to \mathbb{P}(XG.X)$ that maps each relation in R to a subset of events in the partial execution graph. If $x \in \pi(\mathsf{r})$, then we say that x is r-prefix-complete in XG.

Definition 4.7 (Execution prefix). An execution prefix is a triple $\varepsilon = (\rho, XG, \pi)$ where ρ is an initialized run (possibly finite and non-maximal), XG is a partial execution graph over ρ, and π is a prefix-completeness marker over XG. We denote the set of execution prefixes by ExecPre.

Notice that if $\varepsilon = (\rho, XG, \pi)$ is an execution prefix of a maximal run with all reads justified, we can understand it as an execution by dropping π. We often call such execution prefixes simply (complete) executions. The above correspondence highlights the fact that π acts as a ghost state that we only use for reasoning.

We extend the extension order $\sqsubseteq$ from runs to execution prefixes.

Definition 4.8 (Extension order on execution prefixes). We have $(\rho, XG, \pi) \sqsubseteq (\rho', XG', \pi')$ if the following holds:

(i) $\rho \sqsubseteq \rho'$, i.e., ρ is a prefix of ρ'.
(ii) $XG.\mathsf{rf}$ ($XG.\mathsf{co}$) agrees with $XG'.\mathsf{rf}$ ($XG'.\mathsf{co}$) on the common domain $XG.X \subseteq XG'.X$.
(iii) $\pi \subseteq \pi'$ holds point-wise, meaning π' marks more events as prefix-complete than π does.

The requirements on rf and co ensure that *(i)* the partial justifications stay unchanged along extensions and that *(ii)* reads get justified as soon as possible, that is, unjustified reads can only be justified by adding new writes but not by connecting them to already existing ones.

An extension from (ρ, XG, π) to (ρ', XG', π') is called (memory-)fair if it does not change the prefix of marked events, i.e., if $x \in \pi(\mathsf{r})$ then $\textsc{prefix}(XG.\mathsf{r}, x) = \textsc{prefix}(XG'.\mathsf{r}, x)$. Otherwise it is called unfair. We call it proper, if $\rho \sqsubsetneq \rho'$. Consistent, if the resulting execution graph XG' is consistent. And σ-progressing for a thread t, if the underlying extension $\rho \sqsubseteq \rho'$ of runs is σ-progressing for t.

5 Soundness of Recurrence Sets & Lower Semi-continuity

In the previous section we have sketched the proof of Theorem 4.3, which iteratively constructs a sequence of execution prefixes that we claimed to converge to an infinite, consistent, and fair execution. The goal of this section is to prove that the limit execution does indeed satisfy all the desired properties. Towards this, we analyze sequences of executions prefixes and their limits. This requires us to talk about domain theory and continuity.

5.1 Domain-Theoretic Considerations

For each thread t, the set of its runs forms an ω-complete partial order $(\mathsf{Runs}_t, \sqsubseteq)$, meaning that every increasing sequence of thread runs has a limit. Conversely, every (infinite) thread run can be expressed as the supremum of finite runs, i.e., the partial order of runs is algebraic. Similarly, the partial order of program runs $(\mathsf{Runs}, \sqsubseteq)$ also forms an algebraic ω-complete partial order.[2] Our first goal is to show that execution prefixes also form an ω-complete partial order $(\mathsf{ExecPre}, \sqsubseteq)$.

Towards this, we need to make a basic assumption about the event graph function $EG : \mathsf{Runs} \longrightarrow EG(\mathsf{Runs})$, that is satisfied in practice. We assume that the function is Scott-continuous, meaning that it is monotonic and limit-preserving. Monotonicity here means that if $\rho \sqsubseteq \rho'$, then $EG(\rho).\mathsf{r} \subseteq EG(\rho').\mathsf{r}$ holds for all base relations $\mathsf{r} \in \mathbb{R}_B$. Limit-preserving means that if $(\rho)_{i \in \mathbb{N}}$ is an increasing sequence of program runs with limit ρ, then $EG(\rho) = \bigcup_i EG(\rho_i)$. The assumption allows us to prove the following technical lemma, which we use to prove the ω-completeness of $(\mathsf{ExecPre}, \sqsubseteq)$.

Lemma 5.1 (Extension implies set-based inclusion). *Let $(\rho, XG, \pi) \sqsubseteq (\rho', XG', \pi')$. Then $XG \subseteq XG'$ holds component-wise, meaning that $XG.X \subseteq XG'.X$ and for all base relations $r \in \mathbb{R}_B$ (including rf and co) we have $XG.r \subseteq XG'.r$.*

Proof. We have $XG.X = EG(\rho).X \subseteq EG(\rho').X = XG'.X$, because ρ' contains all events of ρ. By monotonicity of the event graph function, we also have for all $\mathsf{r} \in \mathbb{R}_B \setminus \{\mathsf{rf}, \mathsf{co}\}$ that $XG.\mathsf{r} = EG(\rho).\mathsf{r} \subseteq EG(\rho').\mathsf{r} = XG'.\mathsf{r}$. For relations rf and co the inclusion follows directly from the definition of the extension order. □

Lemma 5.2. *$(\mathsf{ExecPre}, \sqsubseteq)$ is an algebraic ω-complete partial order.*

[2]We assume the set of all threads to be countable.

Proof. To show ω-completeness, we need to show that increasing sequences of execution prefixes converge to an execution prefix. Let $(\rho_i, XG_i, \pi_i)_{i\in\mathbb{N}}$ be an increasing sequence. We construct the limit execution (ρ, XG, π) as follows. For the run and the prefix-completeness marker, we choose $\rho = \lim_i \rho_i$ and $\pi = \bigcup_i \pi_i$. For the execution graph XG, we rely on Theorem 5.1: the underlying sequence of execution graphs $(XG_i)_{i\in\mathbb{N}}$ is increasing w.r.t. subset inclusion and therefore has a set-theoretic limit $XG = \bigcup_i XG_i = (\bigcup_i EG(\rho_i), \bigcup_i XG_i.\mathsf{rf}, \bigcup_i XG_i.\mathsf{co})$. By Scott-continuity of the event graph function, we have $XG = (EG(\rho), \bigcup_i XG_i.\mathsf{rf}, \bigcup_i XG_i.\mathsf{co})$, meaning that XG is an execution graph for ρ. It is now easy to see that (ρ, XG, π) is an execution prefix and that it is the least upper bound of the sequence and therefore its limit.

It is also easy to see that $(\mathsf{ExecPre}, \sqsubseteq)$ is algebraic: every infinite execution prefix is the limit of all its finite prefixes. □

We are interested in sequences of execution prefixes that converge to a complete execution where all reads are justified.

Definition 5.3 (Justified sequences). An increasing sequence $(\rho_i, XG_i, \pi_i)_{i\in\mathbb{N}}$ is justified if for every $i \in \mathbb{N}$ and every unjustified read $x \in XG_i.X$ there exists $j > i$ such that x is justified in XG_j.

Lemma 5.4 (Limits of justified sequences). *The limit of a justified sequence has no unjustified reads. Moreover, if the run of the limit is maximal, then the limit is a complete execution.*

Proof. Immediate. □

5.2 Consistency

We now consider justified sequences of consistent execution prefixes and ask the question whether their limits are always consistent. If this was not the case, then our definition of recurrence sets would not be sound. This results in a requirement we impose on the memory model under consideration, which we call consistency-closedness.

Definition 5.5 ((Weakly) consistent sequences and consistency-closedness). Let $(\rho_i, XG_i, \pi)_{i\in\mathbb{N}}$ be a justified sequence. The sequence is called consistent if all of its elements are consistent with the memory model. We say it is weakly consistent, if it contains a consistent subsequence. A memory model *mm* is consistency-closed if limits of weakly consistent sequences are consistent.

If the memory model, understood as a function from execution prefixes to a consistency judgment, is a Scott-continuous function, then it is easily seen to be consistency-closed. This is certainly true for positive memory models, those that do not use negation and difference operators, because the positive operators and the axioms in CAT are Scott-continuous. Interestingly, it may also be true for non-positive memory models. For example, the derived relation not-po = ¬po is non-positive but still monotonic (and continuous): it cannot happen that a po-unrelated pair of events in an execution prefix becomes related by an extension of that prefix.

Unfortunately, there are practical memory models where not all derived relations are monotonically derived. For example, the RISCV memory model defines a relation similar to

$$\texttt{let hb} = [\mathsf{R}]; \mathsf{po}; [\mathsf{R}] \setminus (\mathsf{rf}^{-1}; \mathsf{rf}),$$

which relates two reads in program order if they do not read from the same write. If we consider an execution prefix with two such reads that are not yet justified, then they are related by hb. Now an extension may add a new write that justifies both reads, causing them to become unrelated by hb. Therefore, hb is not monotonically derived and, consequently, is not Scott-continuous. Fortunately, the derivation of hb is still "sufficiently continuous" to be well behaved. Although for any pair of events x and y, $\mathsf{hb}(x, y)$ may change its value in the course of the execution, it will eventually stabilize. We capture this idea (and even more) using a more liberal idea of set-theoretic continuity that does not require monotonicity, namely lower semi-continuity.

For arbitrary sequences of sets, in particular non-monotonic ones, we can define the limit inferior as the set of elements that will, from some moment on, appear in all sets of the sequence. A lower semi-continuous function will, in general, not preserve this limit inferior but instead over-approximate it. To see how this is useful, consider the sequence of hb-relations we see along a sequence of execution prefixes $(\varepsilon)_{i \in \mathbb{N}}$. This sequence might not be monotonic and might even oscillate, but it will have a limit inferior $\liminf_i \mathsf{hb}(\varepsilon_i)$. Furthermore, this limit inferior will be acyclic if the sequence of hb-relations contains infinitely many acyclic relations. Now, if hb is derived in a lower semi-continuous manner, then this relation over-approximates the hb relation of the limit execution which is of the form $\mathsf{hb}(\liminf_i \varepsilon_i)$. This means that the hb-relation of the limit execution has to be acyclic.

Definition 5.6 (Set-theoretic limit inferior). For any sequence of sets $(A_i)_{i \in \mathbb{N}}$, the limit inferior $\liminf_i A_i := \bigcup_i \bigcap_{j \geq i} A_i$ exists.

Definition 5.7 (Lower semi-continuous functions on power sets). Let $f : \mathbb{P}(\mathbb{A}) \rightarrow \mathbb{P}(\mathbb{B})$ be a function between power set lattices (ordered by inclusion), and let $(A_i)_{i \in \mathbb{N}}$ be an arbitrary sequence in $\mathbb{P}(\mathbb{A})$. We say that f is lower semi-continuous if $\liminf f(A_i) \geq f(\liminf A_i)$ holds for all such sequences.

We can extend the continuity definition to functions from ω-complete partial orders such as the extension-ordered execution prefixes into power sets.

Definition 5.8 (Lower semi-continuous functions into power sets). Let $f : \mathbb{A} \rightarrow \mathbb{P}(\mathbb{B})$ be a function from an ω-complete partial order into a power set lattice, and let $(A_i)_{i \in \mathbb{N}}$ be an ω-chain (an increasing sequence) in $\mathbb{A}$. We define $\liminf_i A_i = \bigsqcup_{i \in \mathbb{N}} A_i$ which allows us to reuse the definition of lower semi-continuity as above.

The continuity definition can be applied to memory models when understood as functions from execution prefixes to derived relations (which naturally live in a power set lattice) and the consistency judgment. To be more precise, we can see each (derived) relation r as a function $\mathsf{r} : \varepsilon \mapsto \varepsilon.XG.\mathsf{r}$ and talk about its continuity. Similarly, we can talk about the continuity of the consistency judgment when seeing it as a function $mm_J : \mathsf{ExecPre} \rightarrow \mathbb{B}$ and identifying $(\mathbb{B}, \leq) \simeq (\mathbb{P}(1), \subseteq)$ where $0 \simeq \emptyset$ and $1 = \{1\}$. We say that the memory model (as a whole) is lower semi-continuous or monotonic, if all of its derived relations and its consistency judgment are. We can now prove that if mm_J is lower semi-continuous then the memory model is consistency-closed.

Lemma 5.9 (Consistency-closed if lower semi-continuous). *A memory model mm is consistency-closed if the function mm_J is lower semi-continuous.*

Proof. Let $(\varepsilon_i)_{i \in \mathbb{N}}$ be a weakly consistent sequence of execution prefixes with limit ε. By lower semi-continuity we have $mm_J(\varepsilon) = mm_J(\liminf_i \varepsilon_i) \leq \liminf_i mm_J(\varepsilon_i) = 0$. The last equation holds because the sequence is weakly consistent and, therefore, there is no point from which onward $mm_J(\varepsilon_i)$ takes only value 1. From $mm_J(\varepsilon) \leq 0$ it follows that $mm_J(\varepsilon) = 0$ and so the limit execution ε is consistent. □

There are memory models that are not lower semi-continuous, but their construction requires a particular alternation between negative operators (negation and difference) and projective operators (composition and domain/range projections). For example, consider the memory model given by

$$\mathsf{empty}(\neg((X \times X); \neg\mathsf{dom}(\mathsf{po}); (X \times X))) ,$$

which is satisfied if and only if an execution has po-maximal elements. This property can only fail for infinite executions (if we exclude the existence of special initial writes). Now, if we considered such an inconsistent infinite execution, then the set of its finite prefixes would yield a (consistent) recurrence set that witnesses a non-terminating but inconsistent execution and thus is unsound. Fortunately, as far as we know, no practical memory model makes use of this alternation pattern (the use of negation is quite limited in practical models) and therefore they are all lower semi-continuous and hence consistency-closed. This explains Claim 1.

5.3 Memory Fairness

Let us now consider memory fairness.

Definition 5.10 (Memory-fair sequences). Let mm be a memory model and let $R \subseteq \mathbb{R}$ be a subset of the relation names of the memory model. A justified sequence $(\rho_i, XG_i, \pi_i)_{i\in\mathbb{N}}$ is R-memory-fair if *(i)* for every relation $\mathsf{r} \in R$, every $i \in \mathbb{N}$, and every $x \in XG_i.X$ there exists some $j \geq i$ so that $x \in \pi_j(\mathsf{r})$, and *(ii)* every extension in the sequence is memory-fair, i.e., respects the prefix-completeness marker π_i.

Similar to limits of consistent sequences, the limits of memory-fair sequences are not guaranteed to be memory-fair for arbitrary memory models. Fortunately, lower semi-continuity is also a sufficient condition here. More precisely, we only need that for any event x and any memory-fair relation r, the r-prefix function $\varepsilon \mapsto \textsc{prefix}(\varepsilon.XG.\mathsf{r}, x)$ is lower semi-continuous. This requirement is always satisfied if relation r itself is lower semi-continuous.

Lemma 5.11 (Limits of memory-fair sequences are memory-fair). *Let mm be a memory model and $R \subseteq \mathbb{R}$ be a subset of its relation names. Further, assume that all relations in R are lower semi-continuous. Then limits of R-memory-fair sequences are R-memory-fair.*

Proof. Let $(\rho_i, XG_i, \pi_i)_{i\in\mathbb{N}}$ be an R-memory-fair sequence and let (ρ, XG, π) be its limit. We show that for all $\mathsf{r} \in R$ and all $x \in XG.X$, $\textsc{prefix}(XG.\mathsf{r}, x)$ is finite. The function $\varepsilon \mapsto \textsc{prefix}(\varepsilon.XG.\mathsf{r}, x)$ is lower semi-continuous. Therefore, $\textsc{prefix}(XG.\mathsf{r}, x) \leq \liminf_i \textsc{prefix}(XG_i.\mathsf{r}, x) = \textsc{prefix}(XG_j.\mathsf{r}, x)$ for some j. The inequality holds by lower semi-continuity. The equation holds by memory-fairness of the sequence which guarantees that the r-prefix of x will eventually stabilize at some point j. Since $\textsc{prefix}(XG_j.\mathsf{r}, x)$ is trivially finite, $\textsc{prefix}(XG.\mathsf{r}, x)$ is also finite as desired. □

5.4 Scheduler Fairness

It remains to consider scheduler fairness. Unlike memory fairness, which is defined on the execution graphs, scheduler fairness is defined on the infinite runs of the program. Our first step is to generalize the notion of scheduler fairness to infinite sequences of finite runs that approximate an infinite run.

Definition 5.12 (σ-fair sequences). Let σ be a scheduler fairness function. An increasing sequence of finite runs $(\rho_i)_{i\in\mathbb{N}}$ with infinite limit is called σ-fair if for every i and for every non-terminated thread $t \in \sigma(\rho_i)$ there is $j > i$ such that $\rho_i \sqsubseteq \rho_j$ is σ-progressing for t. This notion naturally lifts to sequences of execution prefixes.

Lemma 5.13. *Limits of σ-fair sequences are σ-fair.*

To prove this lemma, we again rely on lower semi-continuity. Indeed, every scheduler function σ is lower semi-continuous which follows directly from the way we have defined σ on infinite runs in Section 3.1.

Proof. Let $(\rho_i)_{i\in\mathbb{N}}$ be a σ-fair sequence with infinite limit ρ. We show that ρ is σ-fair, meaning for each thread $t \in \sigma(\rho)$, we have that $\rho(t)$ is a maximal run of t. If t has terminated, then it is maximal, so we assume that t has not terminated. By lower semi-continuity of the scheduler function σ, we have $\sigma(\rho) \subseteq \liminf_i \sigma(\rho_i)$, and therefore $t \in \sigma(\rho)$ implies that there is i_0 so that for all $i \geq i_0$ we have $t \in \sigma(\rho_i)$. Scheduler fairness of the sequence says that for each of these $i \geq i_0$ there must be some $j > i$ so that the extension $\rho_i \sqsubseteq \rho_j$ is σ-progressing t. Due to $j \geq i_0$ we know that $t \in \sigma(\rho_j)$, meaning the extension $\rho_i \sqsubseteq \rho_j$ must extend the run of t in order to be σ-progressing. Since there are infinitely many such $i \geq i_0$, the thread must get extended infinitely often. It follows that the thread's run is infinite and thus maximal as desired. □

We are now ready to prove Theorem 4.3, the soundness of recurrence sets.

Proof of Theorem 4.3. Let Rec be a fair recurrence set with scheduler fairness given by σ and memory fairness given by R. Let $\varepsilon \in$ Rec be an execution prefix. We iteratively construct an infinite, justified, and fair sequence starting from ε. Let ε' denote the execution we have constructed so far. Let (i, j) denote the i-th event of the j-th thread. We enumerate through all pairs (i, j) and perform four steps. First, we extend the j-th thread if it is in $\sigma(\varepsilon')$ and has not yet terminated using property *(iii)*. Then, if the i-th event is an unjustified read, we extend the execution prefix to justify this read using property *(ii)*. Lastly, for each relation $\mathsf{r} \in R$, we use property *(iv)* to extend the execution to the point where the i-th event becomes r-prefix-complete. By property *(i)*, we extend once more to obtain a consistent execution prefix. By property *(v)*, all extensions we do are memory-fair. It is important to note that we perform these steps one after another. Fair recurrence sets give us the guarantee that each step can be performed. It is only a consequence of this construction that there is an extension that achieves all properties needed.

It is easy to see that the constructed sequence of execution prefixes is justified and weakly consistent and so converges to a complete and consistent execution. Notice that this limit execution must be infinite, i.e., non-terminating, because we apply property *(i)* infinitely often which is guaranteed to extend the execution sequence properly. Furthermore, by construction the sequence is also R-memory-fair and σ-fair, and so converges to a R-fair and σ-fair execution, using the lemmas established above. This shows that the sequence of execution prefixes converges to an infinite, fair, and consistent execution. □

We remark that the soundness proof does not require lower semi-continuity of the whole memory model, but only of its consistency judgment and those relations that are required to be memory-fair. This means we can slightly weaken the requirements of Theorem 4.3.

6 Abstract Recurrence Sets

Recurrence sets, as defined so far, are infinite in size and thus cannot be represented explicitly. The goal of this section is to define abstract recurrence sets that can yield finite representations for use in automatic verification. Abstract recurrence sets will properly generalize classical state-based recurrence sets, and finite instances will give rise to an execution-based variant of lassos.

Recall that in Section 4 we have argued that classical state-based recurrence sets can be equivalently reformulated as run-based recurrence sets which we then used as a basis to develop our execution-based recurrence sets. The reason for this equivalence is simple: we can extend each state to the set of runs leading to that state, and conversely, we can reduce each run to the state it ends in. Notice how the state here acts as an abstract element that represents a set of runs. With this perspective of "state = set of runs", one may ask how a state transition $s \longrightarrow s'$ relates to the underlying sets of runs represented by s and s'. One might expect that transition $s \longrightarrow s'$ just appends s' to all runs represented by s, but this is not quite true: s' also represents runs that do not visit s immediately before s'. In other words, $s \longrightarrow s'$ does more than just extending the underlying runs by s', it also forgets the past s that lead to s'. We can explain this with a two-step process by thinking of states as (particularly small) infixes of a run: $s \longrightarrow s'$ amounts to first extending the run $s \sqsubseteq s.s'$ and then abstracting away the past $s.s' \subseteq^{\#} s'$. The combined relation $\sqsubseteq^{\#} = \sqsubseteq ; \subseteq^{\#}$, which

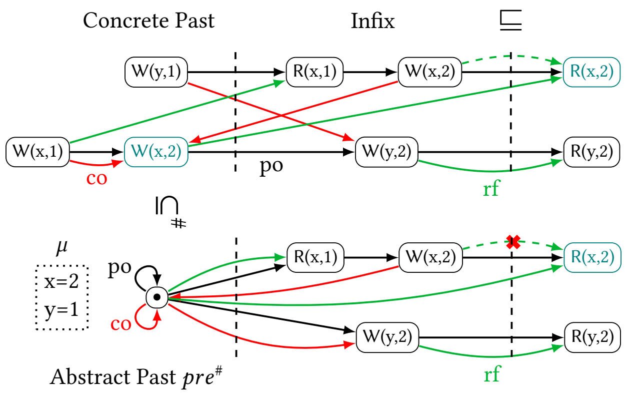

Fig. 3. Concrete execution prefix (top) and a possible abstraction (bottom). The suggested extension on the execution prefix can be simulated on the abstract execution prefix. The dashed rf-edge shows an alternative extension which is consistent with the concrete past but inconsistent with the abstracted past and, hence, cannot be simulated in the abstract.

we call the abstract extension order, then allows us to move from one infix to another like a "sliding window": $\sqsubseteq$ extends the window to the right, $\subseteq^\#$ shrinks it on the left. For abstract recurrence sets, the abstract extension order $\sqsubseteq^\#$ will play the role that state transitions $\rightarrow$ play for classical recurrence sets. The purpose of an abstract recurrence set is now to guarantee us that we can use $\sqsubseteq^\#$ to trace out an infinite execution in such a way that that the infinite execution that results as the limit of this process is fair and consistent.

The key difference between the execution-based setting and the state-based setting will be in how we abstract the past. In the state-based setting, the future of a run is independent of the past of a run: we can forget the past and just remember the most recent state. However, this simple argument fails for execution-based reasoning because the future of a run has to be consistent and memory-fair also with the past of the run. For this reason, $\subseteq^\#$ cannot just forget the past but needs to retain some over-approximate information about it. Similarly, whenever we use $\sqsubseteq$ to extend an execution infix, we need to ensure that this extension is consistent and memory-fair not only with the infix but also with *all possible pasts*. Only then we can guarantee that the infinite execution we trace out via $\sqsubseteq^\# = \subseteq^\#; \sqsubseteq$ will be consistent and memory-fair.

The idea of the abstraction is to collapse a po-prefix of the execution graph into a single abstract event $\bullet$. Edges between the prefix and the infix will be retained as abstract edges of the form $\mathsf{r}(\bullet, x)$ or $\mathsf{r}(x, \bullet)$. The resulting abstract execution graph is an over-approximation of the past that lead to the infix, and if we ensure that extensions are consistent with this over-approximation, then they will be consistent with all matching pasts. Actually, to make this simple argument hold true, we will restrict ourselves to monotonic memory models. That being said, we believe a more refined reasoning can be used to also handle non-monotonic memory models. Lastly, our abstraction cannot avoid collapsing write events that will be read from in future extensions. To handle this, we need to remember the values of co-last write events in a (prefix) memory state μ when collapsing them.

Figure 3 illustrates the abstraction. At the top we have a concrete execution prefix ε, divided into a past and an infix, and a consistent extension of it. The bottom part shows how we abstract the concrete past by collapsing its events into $\bullet$, keeping the edges, and recording the values of co-maximal writes among the collapsed events in μ. On the resulting abstract execution prefix, we can perform the same consistent extension as in the concrete. To see why, consider the highlighted

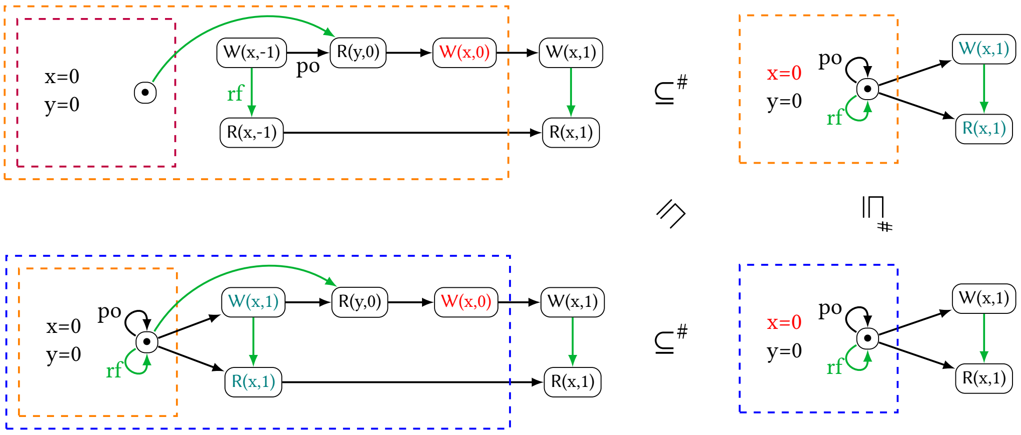

Fig. 4. An abstract recurrence set for the example program from Figure 1 (with coherence edges omitted). There is a repeating abstract execution (top-right and bottom-right) that is related to itself by $\sqsubseteq^\#$, meaning we have found a lasso.

read event R(x,2). We can justify its value from the memory state μ and the consistency of its read-from edge, intuitively, by the fact that it happens strictly after all events in the past. Now consider an alternative extension given by the dashed rf-edge of the read. This extension is consistent in the concrete but it implies a reordering of the read with the highlighted write from the past, which is witnessed by an fr-edge where $\text{fr} = \text{rf}^{-1}; \text{co}$. We cannot justify the consistency of this reordering in the abstract, because it would induce a $(\text{po} \cup \text{fr})$-cycle which is prohibited in most memory models.

As the example suggests, the (concrete) extension order $\sqsubseteq$ on abstract execution prefixes works almost identical to the extension order on concrete execution prefixes, which is the reason why we reuse the same symbol $\sqsubseteq$. The key difference is that we need to be more careful when arguing about consistency and memory fairness of extensions as they need to be consistent and memory-fair w.r.t. all possible pasts. Let us now demonstrate on an example how the three orders, $\subseteq^\#$, $\sqsubseteq$, and the resulting $\sqsubseteq^\#$ interact. Figure 4 shows four abstract executions (with coherence edges omitted) related to the initial example from Figure 1. The top-left abstract execution is essentially an execution prefix of the program with only the initial events abstracted away. We collapse the part that is boxed in orange using $\subseteq^\#$, which gives us the top-right abstract execution. Then, we perform a (concrete) extension $\sqsubseteq$ that appends to each thread the events of a single loop iteration of that thread's loop, giving us the bottom-left abstract execution. We then abstract the part that is boxed in blue using $\subseteq^\#$ again, which leads to the bottom-right abstract execution. We observe that this execution looks identical to the top-right one. We claim that those abstract executions form an abstract recurrence set that gives rise to a *lasso*. We will come back to this in a moment.

We will now sketch the remaining definitions that we need to formulate abstract recurrence sets and state our main theorem. The domain of abstract execution prefixes AbsExecPre is given by elements of the shape $\xi = (\rho, XG, \pi, pre^\#)$, where the first three components form the concrete infix and the last component represents the abstract past. The concrete infix is defined almost like a concrete execution prefix with the following differences: ρ need not be initialized (this is why it is an infix) and XG may have so-called *prefix-justified* read events whose corresponding write event is in the abstract past. We often write $\xi = (\varepsilon, pre^\#)$, highlighting the fact that the first three components resemble a concrete execution. The abstract past has the form $pre^\# = (\mu, XG^\#)$

consisting of the memory state μ and the abstract event $\bullet$ and its edges captured in $XG^{\#}$. Abstract execution prefixes are related to concrete execution prefixes by a concretization function γ that intuitively undoes the collapsing operation shown in Figure 3.

Definition 6.1 (Abstract recurrence sets). Let p be a program, mm a memory model, σ a scheduler fairness function, and R a set of relations. An abstract recurrence set ARec is a non-empty set of abstract execution prefixes of the program p satisfying the following properties for each $\xi \in$ ARec:

(1) ξ has a proper abstract extension $\xi \sqsubseteq^{\#} \xi' \in$ ARec.
(2) For each non-terminated thread $t \in \sigma(\xi)$, there is an extension that σ-progresses t.
(3) For each event $x \in \xi.X$, there is an extension $\xi \sqsubseteq^{\#} \xi' \in$ ARec that abstracts away x.
(4) All of the above extensions are memory-fair.
(5) All of the above extensions are consistent.
(6) There exists $\xi' \in$ ARec such that $\gamma(\xi')$ contains a consistent execution prefix.

Let us explain the main differences between the definitions of abstract recurrence sets and concrete recurrence sets. Property *(3)* formulates a new requirement that every event gets abstracted away eventually. Together with the fact that we will not allow for the abstraction of unjustified reads and events that are not marked as prefix-complete, this property implies properties *(ii)* and *(iv)* of concrete recurrence sets. Property *(5)* now requires consistency of *all* extensions we perform. This stricter requirement is necessary due to the over-approximate nature of the abstraction. Property *(6)* resembles the reachability requirement of state-based recurrence sets: we need to find a consistent execution prefix that reaches some abstract execution in the recurrence set. Property *(2)* about scheduler fairness remains unchanged because it is determined by the infix that we keep track of.

Theorem 6.2 (Abstract recurrence sets are sound). *Let ARec be an abstract recurrence set for program p and monotonic memory model mm. Then there is a fair and non-terminating execution of p that is consistent with mm.*

We can now show that the three different abstract execution prefixes of Figure 4 indeed form an abstract recurrence set. Property *(1)* is satisfied because all executions have proper extensions, including the last one which can extend to the third one and even to itself. Property *(2)* is satisfied for all possible schedulers because all threads are progressing. Property *(3)* is satisfied because every event gets abstracted away eventually. To see this, notice that the two marked events in the top-right execution get abstracted away when extending to itself (bottom-right) along the bottom-left execution. Properties *(4)* and *(5)* also hold, but we will later see why exactly that is. Lastly, Property *(6)* holds because the first execution is clearly reachable: its concretization just adds two initial write events for $x = 0$ and $y = 0$, which yields an execution prefix of the program.

The attentive reader might have noticed that the top-right abstract execution already forms a singleton abstract recurrence set by itself. Precisely such a singleton recurrence set is what we call a *lasso*. It is easy to see that every finite abstract recurrence set must contain a lasso.

We spend the rest of this section making the formalism precise.

6.1 Definitions and Soundness

Definition 6.3 (Abstract prefix). An abstract prefix $pre^{\#} = (\mu, XG^{\#})$ for a concrete infix $\varepsilon = (\rho, XG, \pi)$ is a tuple consisting of a (prefix) memory state μ and an abstract execution graph $XG^{\#}$. The memory state $\mu : \mathbf{Loc} \to \mathbb{Z}$ maps each location $l \in \mathbf{Loc}$ to a value. The abstract execution graph $XG^{\#}$ consists of a single abstract event $\bullet$ and for each relation name $\mathsf{r} \in \mathbb{R}_B$ a set of edges between $\bullet$ and $XG.X$, that is, edges of the form $\mathsf{r}(\bullet, \bullet)$, $\mathsf{r}(\bullet, x)$, and $\mathsf{r}(x, \bullet)$.

In the following, we will omit the word "prefix" and simply talk about concrete executions and abstract executions. We define the abstraction order $\subseteq^{\#} \subseteq \mathsf{AbsExecPre} \times \mathsf{AbsExecPre}$ via a collapsing operation that collapses a prefix into the abstract event $\bullet$.

Definition 6.4 (Collapsing prefixes). Let ξ be an abstract execution and let ρ' be a prefix of $\xi.\rho$. Let $X = \mathsf{Ev}(\rho')$ be the set of events in the prefix. We construct a new abstract execution ξ' from ξ as follows. We collapse X into the abstract prefix by mapping those events to $\bullet$ while preserving edges (homomorphic mapping). This yields $\xi'.XG^{\#}$. If we collapse a store $w(l, v)$ that is co-maximal among the collapsed events and the past, i.e., there are no $\mathsf{co}(w, X \cup \{\bullet\})$-edges, then we update the prefix memory state to $\xi'.\mu[l \leftarrow v]$.

Definition 6.5 (Abstraction order). We have $\xi \subseteq^{\#} \xi'$ if the former can be collapsed to an ξ'' such that $\xi''.\varepsilon = \xi'.\varepsilon$, $\xi''.\mu = \xi'.\mu$, and for all base relation $\mathsf{r} \in \mathbb{R}_B$ we have $\xi''.XG^{\#}.\mathsf{r} \subseteq \xi'.XG^{\#}.\mathsf{r}$. For co and rf we require the inclusion to be satisfied with equality. Furthermore, we disallow collapsing of unjustified reads and events not marked as prefix-complete w.r.t. every memory-fair relation.

We will see in a moment why the relaxation to subset inclusion of the abstract execution graphs is helpful, and why we require exactness for coherence.

For any concrete execution $\varepsilon \in \mathsf{ExecPre}$, we define a canonical abstract execution $\xi(\varepsilon)$ by collapsing all initialization events in ε. This gives rise to the concretization function

$$\gamma : \mathsf{AbsExecPre} \longrightarrow \mathbb{P}(\mathsf{ExecPre}),\ \gamma(\xi) = \{\varepsilon \in \mathsf{ExecPre} \mid \xi(\varepsilon) \subseteq^{\#} \xi\}\ .$$

The concretization essentially undoes the collapsing operation, but it does not need to do so exactly: abstract edges (other than co and rf) like $\mathsf{po}(\bullet, x)$ are treated like upper bounds, meaning they do not need to be witnessed in the concretization. In other words, the abstract past represents an upper bound (a worst-case) of the events and relations of the past. However, it is important that co is treated exactly in order to justify our collapsing operation: if we have an edge $\mathsf{co}(x, \bullet)$ and x gets collapsed into $\bullet$, then we know that, within the resulting past, x must have a co-successor and, hence, cannot be co-maximal inside the past. This is why we do not need to update μ in this case.

From the definition of the concretization, it is easy to see that $\xi \subseteq^{\#} \xi'$ implies $\gamma(\xi) \subseteq \gamma(\xi')$, meaning that more abstract executions represent larger sets of concrete executions. This is why we denote the abstraction order $\subseteq^{\#}$ by the set inclusion symbol. Also notice that the concretization will contain, in general, many inconsistent executions. This is no problem because all we need to ensure is the existence of a single consistent execution, which our abstract recurrence sets will do.

We now define the extension order $\sqsubseteq$ on abstract executions, i.e., how we extend the concrete infix of abstract executions. The idea of an extension $\xi \sqsubseteq \xi'$ in the abstract is that it can be replayed on all underlying concrete executions, i.e., it witnesses the existence of (canonical) extensions in the concrete. More precisely, on each underlying concrete execution $\varepsilon \in \gamma(\xi)$ we can append the events $(\xi'.X \setminus \xi.X)$ and add rf and co-edges in a canonical way to obtain an execution $\varepsilon' \in \gamma(\xi')$.

Naively, we would like to define $(\varepsilon, pre^{\#}) \sqsubseteq (\varepsilon', pre'^{\#})$ iff $\varepsilon \sqsubseteq \varepsilon'$ and $pre^{\#} = pre'^{\#}$. However, we need to record edges between newly appended events and the abstract event $\bullet$, meaning we have to update $pre^{\#}$. We sketch this update for the common base relations. If we append a prefix-justified read $r = \mathsf{R}(l, v)$, then we record $\mathsf{rf}(\bullet, r)$ and require that the observed value matches with the prefix memory state, i.e., $v = \mu(l)$. This abstract edge can be realized in every concretization of the past, since the concretization will contain a write event with value $\mu(l)$. If we append a store w then we record $\mathsf{co}(\bullet, w)$ which is realized by the initialization events that are always initial in the coherence order. We will never record $\mathsf{co}(w, \bullet)$ because this edge might not be realizable: if the abstract past consists of only the initialization writes then w cannot possibly be co-before any of them. For any appended event x, we unconditionally record $\mathsf{po}(\bullet, x)$. We can do so because of the aforementioned relaxation to upper bounds: the po-edge does not have to be witnessed. We do similar updates for all other existing base relations $\mathsf{r} \in \mathbb{R}_B$. Theoretically, we could always record $\mathsf{r}(\bullet, x)$ and $\mathsf{r}(x, \bullet)$, but this will be too imprecise for our use-case.

Lemma 6.6 (Soundness of extensions of abstract executions). *If $\xi \sqsubseteq \xi'$, then for all $\varepsilon \in \gamma(\xi)$ there exists a canonical extension $\varepsilon \sqsubseteq \varepsilon' \in \gamma(\xi')$.*

Based on the above correspondence, we can directly lift the concepts of proper, scheduler-progressing, and memory-fair extensions from executions to abstract executions: $\xi \sqsubseteq \xi'$ is said to be proper (scheduler-progressing/memory-fair) if for all $\varepsilon \in \gamma(\xi)$ its canonical extension is proper (scheduler-progressing/memory-fair). For consistency, we relativize the lifting and say that $\xi \sqsubseteq \xi'$ is consistent if for all *consistent* $\varepsilon \in \gamma(\xi)$ its canonical extension is again consistent. All of the above notions also apply to abstract extensions (recall that $\sqsubseteq^{\#} = \sqsubseteq; \subseteq^{\#}$): $\xi \sqsubseteq^{\#} \xi'$ has one of the above properties if $\xi \sqsubseteq \xi'' \subseteq^{\#} \xi'$ and $\xi \sqsubseteq \xi''$ has said property. We are ready to proof our main theorem.

Proof of Theorem 6.2. Let ARec be an abstract recurrence set. We show that $\mathrm{Rec} := \{\varepsilon \in \bigcup \gamma(\mathrm{ARec}) \mid \varepsilon \text{ is consistent}\}$ is a fair recurrence set. By property *(6)*, we have $\mathrm{Rec} \neq \emptyset$. Further, observe that all executions in Rec are consistent. Now, let $\varepsilon \in \gamma(\xi) \subseteq \mathrm{Rec}$ be a concrete execution prefix in the set, then we need to show that it has extensions satisfying properties *(i)*-*(iv)*.

By property *(1)* together with property *(5)* there is a proper and consistent abstract extension $\xi \sqsubseteq^{\#} \xi' \in \mathrm{ARec}$, meaning that $\xi \sqsubseteq \xi'' \subseteq^{\#} \xi'$ and $\xi \sqsubseteq \xi''$ is proper and consistent. By Theorem 6.6, there exists a canonical extension $\varepsilon \sqsubseteq \varepsilon' \in \gamma(\xi')$ that is proper and consistent. It follows that ε' is consistent and hence contained in Rec as desired. To show property *(ii)*, let x be an unjustified read in $\varepsilon.X$. By properties *(3)* and *(5)* there is a consistent extension $\xi \sqsubseteq^{\#} \xi''$ that abstracts away x, but we cannot abstract unjustified reads, meaning this extension must also justify the read before abstracting it. Again, by Theorem 6.6, there is a consistent extension $\varepsilon \sqsubseteq \varepsilon'' \in \gamma(\xi'')$ that justifies the read. Observe that ε'' is consistent and so contained in Rec. By a similar argument, Property *(iv)* holds because for every event that is not marked as prefix-complete there is an extension that abstracts away this event, but we only allow abstraction of marked events, therefore that extension must mark the event as prefix-complete. Property *(iii)* immediately follows from property *(2)*. Lastly, property *(v)* follows from property *(4)*. □

6.2 Checking Properties of Abstract Extensions Algorithmically

It remains to show how we can algorithmically check that abstract extensions satisfy the properties required by abstract recurrence sets without explicitly reasoning over all underlying concrete executions. This is obvious for properness and scheduler-progress, but it is not obvious for memory fairness and consistency. We focus on memory fairness first.

Consider an extension $\xi \sqsubseteq \xi'$ and memory fairness with respect to $\mathsf{fr} = \mathsf{rf}^{-1}; \mathsf{co}$. The basic idea is to ensure that the extension introduces no fr-paths from newly added events into the abstract event $\bullet$.[3] The naive way to ensure this is as follows: we combine $\xi'.XG$ and $\xi'.XG^{\#}$ into a single execution graph and compute fr over it. Then we check if some newly added event has an fr-edge into $\bullet$. However, this will (almost) always be the case if we add a prefix-justified read r, because we will have $\mathsf{rf}(\bullet, r)$ and therefore $\mathsf{rf}^{-1}(r, \bullet)$, but also $\mathsf{co}(\bullet, \bullet)$. Those two edges combine to $\mathsf{fr}(r, \bullet)$.

We illustrate this issue in Figure 5 which elaborates on the extension performed in Figure 4. The top execution corresponds to the third (bottom-left) execution of Figure 4 but extended with some previously omitted co-edges and their induced fr-edges. Among the newly appended events, R(y,0) has a fr-edge that goes into the past seemingly violating memory fairness (and consistency). However, that fr-edge is not realized in the corresponding canonical extension of any underlying concrete execution. The reason is that we justify R(y,0) by letting it read from the

[3] We assume that within the concrete infix no event is marked as prefix-complete.

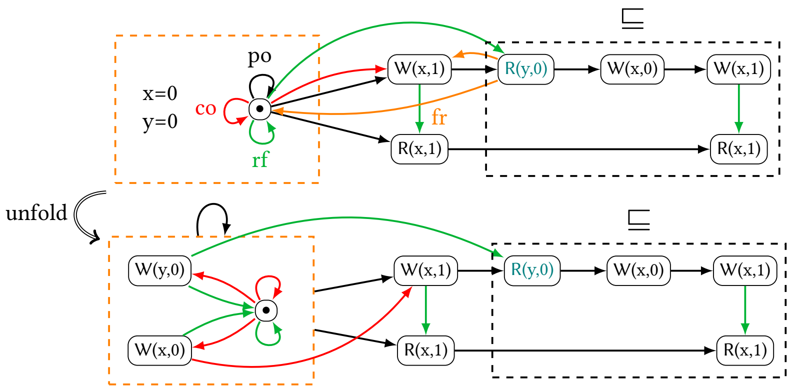

Fig. 5. Checking the memory fairness and consistency of the ⊑-extension from Figure 4. The top execution graph has problematic fr-edges that disappear after unfolding the past (bottom execution graph). The po-edges of the unfolded past mean that every event inside the past has such an edge.

co-maximal write in the past rather than *any write*. That write has no co-successors within the past. Consequently, no fr-edge from r = R(y,0) leads into the past; the abstract fr$(r, \bullet)$-edge is a spurious over-approximation. A similar argument can be made to show that the other fr-edge of R(y,0) is spurious as well. Although this edge might not violate memory fairness, it would certainly violate consistency in most memory models.

We recover from this imprecision by partially unfolding/concretizing the abstract past on demand: for each memory location $l \in \mathbf{Loc}$ accessed by a newly added prefix-justified read r, we add a new write event $w = \mathrm{W}(l, \mu(l))$ that represents the co-maximal write of the past and let r read from it, i.e., we realize the abstract rf$(\bullet, r)$-edge by rf(w, r). The new write events have the same edges as $\bullet$ except for co-edges and rf-edges, meaning that with respect to all other edges $\bullet$ and the new events behave equivalently. Notice that prefix-justified reads that existed already in the non-extended execution ξ may end up having two rf-edges, one from the abstract event and one from the new write event. The reason is that we do not know if they read co-maximal from the past [4]

Coming back to our example, the unfolding can be seen in the bottom of Figure 5. Notice how the problematic fr-edges are gone after unfolding: the extension is memory-fair and consistent.

In general, there is another problem we have to consider: what if the newly added events do not themselves have fr-edges into the past but cause already existing events to get new such edges? Indeed, suppose that we added a new write event w that we use to justify a yet-unjustified read r in the infix with edge rf(w, r), and for the sake of argument suppose that we also added a co$(w, \bullet)$-edge (we do not allow this in our definitions), then we would end up adding an fr$(r, \bullet)$-edge from an already existing event. We cannot permit this. Even worse, in the general case of r-memory-fairness, we might add an r-edge from infix into the past that already existed beforehand, i.e., we add a new witness for an already existing abstract edge. We cannot permit this either. What this means is that we need to reason not only about the edges of the combined graph but also how they can be derived: we cannot admit a derivation of an r-edge into the past that involves a newly added event/edge of the extension.

Fortunately, we can check this algorithmically by instrumenting the memory model *mm* to keep track of how it derives edges, in particular, whether a derivation involves a newly added event.

[4]Only if the value of the read does not match with μ, we know that it cannot have read co-maximal from the past.

We use the same trick to check consistency: we evaluate the instrumented memory model on the combined and partially unfolded execution graph and check if there is a consistency violation involving a new event. The idea of the instrumentation is to collect the newly appended events $\xi'.X \setminus \xi.X$ into a new base set S and define for each relation $\mathsf{r} \in \mathbb{R}$ an instrumented version r' that captures the subset of r-edges that have at least one derivation involving some event from S. The instrumentation is defined inductively along the structure of the CAT language:

$$\begin{array}{rlrlrl}
\mathsf{br}' &= [\mathsf{S}];\mathsf{br} \cup \mathsf{br};[\mathsf{S}] & (\mathsf{a};\mathsf{b})' &= \mathsf{a}';\mathsf{b} \cup \mathsf{a};\mathsf{b}' & \mathbf{empty}(\mathsf{r})' &= \mathbf{empty}(\mathsf{r}') \\
\mathsf{bs}' &= \mathsf{bs} \cap \mathsf{S} & (\mathsf{a}^{+})' &= \mathsf{a}^{*};\mathsf{a}';\mathsf{a}^{*} & \mathbf{irreflexive}(\mathsf{r})' &= \mathbf{irreflexive}(\mathsf{r}') \\
(\mathsf{a} \cup \mathsf{b})' &= \mathsf{a}' \cup \mathsf{b}' & (\mathsf{a}^{-1})' &= (\mathsf{a}')^{-1} & \mathbf{acyclic}(\mathsf{r})' &= \mathbf{irreflexive}((\mathsf{r}^{+})') \\
(\mathsf{a} \cap \mathsf{b})' &= (\mathsf{a}' \cap \mathsf{b}) \cup (\mathsf{a} \cap \mathsf{b}') & [\mathsf{a}]' &= [\mathsf{a}'] & (\texttt{let}\ rn := \mathsf{r})' &= \texttt{let}\ rn := \mathsf{r} \land \texttt{let}\ rn' := \mathsf{r}' \\
(\mathsf{a} \setminus \mathsf{b})' &= (\mathsf{a}' \setminus \mathsf{b}) \cup (\mathsf{a} \setminus \mathsf{b}') & (\mathsf{a} \times \mathsf{b})' &= (\mathsf{a}' \times \mathsf{b}) \cup (\mathsf{a} \times \mathsf{b}') & (mm \land mm)' &= mm' \land mm'
\end{array}$$

Lemma 6.7. *Let $\xi \sqsubseteq \xi'$ be an extension, G the corresponding combined and partially unfolded execution graph of ξ', and mm' the instrumented memory model as above. If G is consistent with mm', then $\xi \sqsubseteq \xi'$ is a consistent extension. If G has no $(r^{*};r';r^{*})$-edges into events $\pi(r) \cup \{\bullet\}$, then $\xi \sqsubseteq \xi'$ is r-memory-fair.*

We need to address one important point to justify the above construction. Notice that none of the above arguments would suffice if an extension could enlarge the base relations between events in the past, because that could cause consistency and/or fairness violations that we cannot observe in the abstraction. The assumption that base relations are derived monotonically from the run is not sufficient to exclude this case. Fortunately, all practical base relations r satisfy a much stronger property: $(x, y) \in \varepsilon'.\mathsf{r}$ iff $(x, y) \in \varepsilon.\mathsf{r}$ for all $\varepsilon \sqsubseteq \varepsilon'$ with $x, y \in \varepsilon.X$. What this means is that the base relations that hold between two events are determined the first time they appear together in the execution, and will not change in any extension of the execution. This property has a few nice consequences: *(i)* base relations within the past and between existing events are unaffected by extensions and *(ii)* the negation of base relations is monotonic. Point *(i)* is what makes our arguments sound: it guarantees that all memory fairness and consistency violations an extension introduces must involve the newly added events. Point *(ii)* means that, for example, po and its negation $\neg\mathsf{po}$ grow monotonically along increasing execution sequences. From this it follows that most memory models, even those relying on negations and differences, are in fact monotonic.

We make a few final observations and remarks about abstract recurrence sets. First, our construction guarantees to never add co-edges into the past, and therefore it can only witness co-fair non-termination. Similarly, for each acyclicity constraint $\mathsf{acyclic}(\mathsf{r})$, we often incidentally guarantee r-fairness as well. The reason is that r-paths into the abstract past are almost always prohibited by the consistency checks we perform. Consequently, the abstraction is only suitable when assuming memory fairness properties that align with the ordering constraints of the memory model. That being said, we believe a less aggressive abstraction of the past can be used to also handle weaker notions of memory fairness.

Secondly, the abstraction we presented is a pure shape abstraction: we abstract nodes and edges but keep data values concrete. It is easy to imagine that this can be combined with data abstractions. For example, data values in events and/or the past could be described symbolically possibly by predicates like $x > 0$. By doing so, we might obtain finite representations even for non-terminating executions that do not have a strictly repeating data flow.

6.3 Recovering Classical State-Based Recurrence Sets

We now demonstrate how abstract recurrence sets properly generalize state-based ones. Consider an abstract execution $\xi = (\rho, XG, \pi, \mu, XG^{\#})$ whose concrete infix is minimal, meaning $\rho = c$ is a single configuration (local state per thread) without any transitions, XG and π are empty, and $XG^{\#}$ is just a single abstract event $\bullet$ with self-loops. Therefore, the minimal abstract execution is effectively represented by a tuple $\sigma = (c, \mu)$, which is precisely a classical state in the interleaving semantics of concurrent programs (local state per thread c + shared global state μ). Over these "states" (minimal abstract executions), we can consider a subset of abstract extensions that append a single event and immediately abstract it away to mimic classical single-step transitions. Write events appended in this way update μ with the value of the write. Read events appended this way always read the most recent value from μ and update the local state in c accordingly.

Now consider a set of such states with an unfair scheduler and the sequential consistency (SC) memory model defined by acyclic(hb) and `let hb` := po ∪ rf ∪ co ∪ fr. We check under what conditions this set forms an abstract recurrence sets. Property *(2)* is trivially satisfied by the unfair scheduler, property *(3)* is satisfied because we abstract all events immediately, properties *(4)* and *(5)* are satisfied for SC because each extension only adds hb-edges that go forwards. Only properties *(1)* and *(6)* are not trivially satisfied. The former corresponds to the existence of transitions inside the recurrence set and the latter corresponds to reachability, precisely the two conditions of classical state-based recurrence sets. With this argumentation, our abstract execution-based recurrence sets are a proper generalization of the classical ones.

7 Application to Automatic Verification

We now show how to apply our theory to automatically find non-termination issues in real programs by looking for lassos.

7.1 From Abstract Lassos to Concrete Lassos

Recall that we defined a lasso to be an abstract recurrence set consisting of a single abstract execution prefix ξ. Such a lasso can be witnessed by a single *concrete* execution prefix $\varepsilon \in \xi$, which allows us to employ any execution-graph-based reachability checking technique to find them [21, 38]. Let us first demonstrate lassos in the classical setting.

In the classical state-based setting, a lasso is a single run $\rho = \rho_{stem}.s.\rho_{loop}.s$ that repeats some state s [15]. The state s forms a singleton recurrence set (which we also call a lasso) with respect to the transitively-closed transition relation $\rightarrow^{+}$. The stem of the run $\rho_{stem}.s$ witnesses the reachability of the recurrence set, whereas the loop $s.\rho_{loop}.s$ witnesses the repeatability $s \rightarrow^{+} s$. We can do the same in the execution-based setting by looking for a single concrete execution prefix ε that consistently and fairly repeats an infix. Let us explain.

Consider a consistent execution ε' with a run that can be partitioned into $\varepsilon'.\rho = \rho_{stem}{\cdot}\rho_{inf}{\cdot}\rho_{loop}{\cdot}\rho_{inf}$ and its prefix $\varepsilon = \varepsilon' \downarrow \rho_{stem}{\cdot}\rho_{inf}$ obtained by restricting the execution to the first occurrence of the infix. We can construct abstract executions ξ and ξ' from ε resp. ε' by collapsing ρ_{stem} in both. We then check if $\xi \sqsubseteq \xi'$ holds and whether the extension is consistent and fair. This checks if an abstract execution with run infix ρ_{inf} consistently and fairly extends to an abstract execution with run infix $\rho_{inf}{\cdot}\rho_{loop}{\cdot}\rho_{inf}$. If further $\xi' \subseteq^{\#} \xi$ holds, i.e., abstracting away the $\rho_{inf}{\cdot}\rho_{loop}$-part leads to a repeating past, we have found a lasso: $\xi \sqsubseteq \xi' \subseteq^{\#} \xi$ implies $\xi \sqsubseteq^{\#} \xi$. Since ε' induces an (abstract) lasso, we also call ε' a (concrete) lasso. It should now be easy to see that the concrete execution prefix shown in our initial example Figure 1 is a lasso indeed. The corresponding abstract lasso is given by the the top-right (or bottom-right) execution in Figure 4.

7.2 Finding Concrete Lassos using SMT

We opted for an SMT-based technique to find lassos using DARTAGNAN [36–38]. DARTAGNAN already encodes all consistent program executions up to a user-specified unrolling bound of the loops. All we need to do is encode whether such an execution induces a lasso. We sketch the encoding.

For each execution, we let the SMT solver guess a partitioning into four parts, the events before the infix (the stem/the past), the events in the infix, and the extension of which a suffix repeats the infix. To this end, we introduce Boolean variables that indicate whether a loop iteration belongs to the infix resp. the suffix. Moreover, we have a Boolean variable that indicates whether two loop iterations are matching, as defined below. Now we look for an execution where each thread that fails to terminate (reaches the unrolling bound) and is subject to scheduler fairness has a loop iteration in the infix with a matching partner in the suffix. The matching constraints are:

- matching loop iterations have to emit the same events with the same values (repeating control and data flow),
- if there are rf/co-edges within the infix, then also the suffix has to have the same edges (repeating justification).

Lastly, we need to encode the consistency and memory fairness checks. We could use the instrumented memory model *mm′* from Theorem 6.7 and encode it just like Dartagnan encodes the non-instrumented model *mm*. However, we chose to approximate the checks instead. Let us explain.

Since we only encode {co, fr}-fairness, we do not need the full instrumented memory model to check memory fairness. Instead, we just require writes in the suffix to be co-after writes in the prefix (co-fairness), and that reads in the suffix only read from writes in the prefix if they are *globally* co-maximal (fr-fairness). We simply omit the consistency check of the extension. However, note that we still check consistency of the execution as a whole. This might give spurious lassos in theory, but it never does so in practice. The reasons are twofold. On the one hand, restricting co and rf between suffix and prefix already heavily restricts derived relations like hb between suffix and infix, oftentimes sufficiently enough to pass the omitted consistency check of the extension. On the other hand, even if the check had failed, the extension might still be consistent for the concrete past we have at hand (recall that the check reasons about all possible pasts). Indeed, the fact that we found an execution that could consistently repeat its infix twice already indicates that further repetitions will likely stay consistent.

The advantage of omitting the consistency check of the extension is that, apart from giving smaller encodings, it works out-of-the-box with Dartagnan's lazy solving approach that uses its own theory solver for consistency and hence can avoid encoding the memory model [17].

8 Evaluation

Our theory reasons about weak consistency, memory fairness, scheduler (un)fairness, and side-effects. We set up several experiments to show-case the need for all those features. It should be noted, however, that we have not found a single benchmark that requires reasoning about all aspects simultaneously.

We implemented the encoding from Section 7 in Dartagnan[5]. We also implemented the theory from [22], which allows us to prove termination of programs where all spin loops are side-effect free. Dartagnan reasons about litmus tests written in a pseudo-assembly format, real programs written either in C or several shading languages (including Slang, OpenCL and Hlsl via Spir-V), weak consistency models written in CAT, and a predefined set of scheduler functions taken from [46].

8.1 Validation

We performed three experiments to validate our theory and implementation. For this, we use *(i)* synthetic benchmarks that show some complex patterns handled by our theory, *(ii)* several litmus tests previously used to specify GPU workgroup forward progress models [46], and *(iii)* concurrency benchmarks from the Software Verification Competition (SV-COMP [9]).

[5]Our implementation is available starting from version 4.3.0.

Table 1. Validating Dartagnan on synthetic non-termination benchmarks (left) and litmus tests w.r.t. different forward progress models (right).

Benchmark	Terminates	Time
asymmetric	✗	0.7s
complex	✗	1.0s
oscillating	✗	0.5s
weak	✗	0.8s
xchg	✗	0.6s
zero_effect	✗	6.9s

Scheduler	Pass	Fail	Unknown	Error
Fair	104	156	217	6
Obe	8	453	16	6
Hsa	42	388	47	6
Hsa+Obe	46	367	64	6
Lobe	46	356	75	6

Synthetic Benchmarks. Since none of the real programs in Sections 8.2 and 8.3 cover all complex scenarios our theory supports, we created six programs with non-terminating loops on different threads. They require reasoning about memory fairness, weak memory, and side-effects:

(i) *asymmetric:* until `y==1`, loop-1 alternates between assigning `x=0` and `x=1`. Loop-2 checks if `x==0 || x==1`, and if so, terminates and sets `y=1`. If between each check the value gets flipped, the program fails to terminate; this requires two iterations of loop-1 for each iteration of loop-2.

(ii) *complex:* three loops in different threads interfere with each other. Any pair of loops would terminate, but all three together do not.

(iii) *oscillating:* at each iteration, one loop changes a memory value first to zero and then to one, i.e., the value oscillates. Another loop always observes the same value, and thus it continues spinning. This is the test shown in Figure 1.

(iv) *weak:* a message passing pattern repeats until success. This test fails to terminate on memory models that require barriers to make message passing work, i.e., anything weaker than TSO.

(v) *xchg:* a loop that tries to acquire a taken lock using side-effectful atomic exchange operations.

(vi) *zero_effect:* Similar to *xchg* but uses fetch-add operations that cancel each other.

For these benchmarks, we used IMM as the memory model [35]. Table 1 (left) shows that Dartagnan finds all non-termination bugs.

Forward Progress Litmus Tests. We used 483 litmus tests that were previously used to study different forward progress models for GPU concurrency [46]. The tests do not make use of weak concurrency, and thus the results are independent of the memory model. However, since the tests make use of the memory hierarchy found on GPUs, we used the Vulkan memory model [47]. Many of these tests contain loops with side-effects, meaning we cannot always prove termination. However, we can find all non-terminating instances. These benchmarks require handing of scheduler (un)fairness, memory fairness, and side-effects.

The results are given in Table 1 (right). The Error column contains programs which Dartagnan cannot unroll since they contain loops that cannot be normalized. For all results in the Pass and Fail columns, Dartagnan agrees with the results from [46]. Column Unknown shows those terminating tests that have side-effects.

SV-COMP Benchmarks. Although the competition has categories for concurrency and termination, there is not yet a category that requires reasoning about both. However, a proposal to introduce such a category adds termination results for 183 concurrency benchmarks [41]. The expected results were generated by the UAutomizer tool [12] and complemented by some benchmarks that were manually labeled. These benchmarks require reasoning about memory fairness, side-effects, and as we will show below, scheduler (un)fairness.

Table 2. Detecting non-termination under weak memory on real C programs.

Benchmark	GenMC (M)	GenMC (A)	Dartagnan (A)
arraylock	✓ (0.1s)	✓ (0.1s)	✓ (1.7s / B=4)
caslock	✓ (0.2s)	🕒	✓ (2.0s / B=4)
clhlock	✓ (0.1s)	🕒	✓ (1.5s / B=4)
cnalock	✗ (0.2s)	✗ (0.2s)	✗ (11.9s / B=2)
hclhlock	✗ (0.5s)	✗ (0.5s)	✗ (1m 58s / B=2)
hemlock	✓ (0.2s)	🕒	✓ (2.3s / B=4)
hmcslock	✗ (0.3s)	✗ (0.3s)	✗ (44.3s / B=8)
mcslock	✗ (0.1s)	🕒	✗ (1.2s / B=2)
rec_mcslock	✗ (0.1s)	🕒	✗ (2.3s / B=4)
rec_seqlock	✓ (1.3s)	✓ (0.6s)	✓ (6.7s / B=4)

Benchmark	GenMC (M)	GenMC (A)	Dartagnan (A)
rec_spinlock	✓ (0.4s)	🕒	✓ (2.9s / B=4)
rec_ticketlock	✓ (0.1s)	🕒	✓ (2.7s / B=4)
rwlock	✓ (0.5s)	🕒	✓ (20.8s / B=4)
semaphore	✓ (0.1s)	✓ (0.1s)	✓ (21.6s / B=4)
seqcount	✓ (0.1s)	✓ (0.1s)	✓ (0.6s / B=4)
seqlock	✓ (0.3s)	✓ (0.2s)	✓ (1.9s / B=4)
ticketlock	✓ (0.1s)	🕒	✓ (1.3s / B=4)
ttaslock	✓ (0.1s)	🕒	✓ (2.1s / B=4)
twalock	✗ (0.3s)	🕒	✗ (1.7s / B=2)

From the 143 benchmarks that are claimed to terminate, Dartagnan can prove termination for 43 of them (since all spin loops are side-effect free), throws an error for 2, and reaches a 15 minutes timeout for the remaining 98. For the 40 benchmarks that are claimed not to terminate, Dartagnan reports non-termination for 22 of them, throws an error for 6, and reports a timeout for 9. Interestingly, it reports that 3 of these benchmarks terminate. The developers of UAutomizer acknowledged that this is because their tool does not assume any kind of fairness. If we enable the unfair scheduler in Dartagnan, it also reports non-termination for the 3 of them.

8.2 Termination of Real C Programs

This section compares Dartagnan and GenMC[6] [21] on several synchronization primitives from the Libvsync library [39]. GenMC is the only other tool that reasons about memory models and termination. However, it only handles side-effect free loops. Libvsync benchmarks are correct (including termination) even in the presence of weak memory models. However, it has previously been shown that some barriers can be relaxed without violating safety, i.e., mutual exclusion [34]. This means that some memory-model-related bugs only manifest as non-termination bugs and so we need a theory that can reason about both. We relaxed barriers in such a way that mutual exclusion is still preserved. However, such relaxations introduce non-termination bugs in six of the benchmarks. Since by default Libvsync relies on manual annotation of spin loops, we run GenMC both using these annotations and relying on its automatic detection. For Dartagnan we rely purely on its automatic spin loop detection. We use a 15 min timeout for each verification run.

Both tools can only prove correctness if executions are bounded. The automatic spin loop detection of GenMC fails in more than half of the benchmarks. Not being able to bound the executions, the tool reaches a timeout. When manual annotations are used, GenMC finds all non-termination bugs and proves the remaining benchmarks correct. Dartagnan requires the user to explicitly set the unrolling bound. We used bounds large enough to find violations or prove correctness. The used bound is given next to the solving time. Dartagnan finds all non-termination bugs and proves all remaining benchmarks correct. Notice that when both tools terminate, GenMC is much faster than Dartagnan.

8.3 Termination of Real GPU Computing Kernels

Prefixsum is a core building block for GPU algorithms [2, 10, 42]. Given an input array, it returns an array where the n-th element is the sum of the preceding subsequence of elements. In GPUs, the computation is split across different sets of threads (called workgroups). Each workgroup performs the prefixsum of its portion of the input, waits for the results of the *preceding* workgroup so it

[6]Version v0.11.0 (commit #b03f01e). Versions v0.12.0 to v0.13.1 (the newest at the time of publication) report wrong results on *seqlock* and *rec_seqlock*.

can incorporate it into its own result, and it then forwards the result to the *next* workgroup. The waiting and forwarding is a message passing pattern across workgroups.

Depending on how *preceding* and *next* are computed, the code might hang. The problem is that if there are not enough resources to allocate all workgroups on hardware, the scheduler will split them into several batches. If a workgroup in one batch waits on a workgroup of a not-yet executed batch, the program will hang if resources are never freed. This is what happens on some current GPUs. While it is known that this algorithm is not portable across different GPUs due to the lack of forward progress guarantees [24, 43], to the best of our knowledge, no tool has been able to automatically prove this until now.

What makes this problem particularly challenging, is that it requires to reason about progress models, and termination[7]. We used Dartagnan to verify three different implementations of prefix-sum. One developed at UCSC using OpenCL [13, 27], one developed as part of the Vello graphics rendering engine using Hlsl [25], and one developed by us using Slang. All three implementations follow the state-of-the-art *decoupled look-back* [32]. The benchmarks are configured to have three workgroups (to force executing the look-back logic) having two threads each (to force intra-workgroup synchronization via control barriers[8]).

The original implementation of prefixsum uses the id of the workgroups to decide how they wait for each other. Table 3 shows that for this implementation, the forward progress guarantees given by HSA are enough to guarantee termination, but those of OBE are not. An alternative approach is to make each workgroup obtain a ticket using an atomic increment before performing the core logic of the algorithm. In this case, the waiting is on a workgroup that has already performed at least one execution step (otherwise it would not have obtained a smaller ticket). The OBE model guarantees that if a workgroup performed at least one execution step, then it experiences fair scheduling and thus it will eventually forward its result. Table 3 shows that this is enough to guarantee termination of decoupled look-back. While it has been empirically shown that most current GPUs implement OBE, at least some Apple devices still violate this model [46].

Interestingly, the Vello implementation terminates even under an unfair scheduler. This implementation uses a work-stealing method called *scalar fallback*. At each spinning iteration, it processes one element corresponding to the workgroup it waits for. Since the number of such elements is bounded, Table 3 shows the approach terminates even if no forward progress guarantees are given (Dartagnan requires a larger unrolling bound which considerably increases the verification time). This algorithm has two limitations: the fallback is performed sequentially by a single thread and the result is never posted to device memory, forcing every blocked workgroup to recompute the fallback. *Decoupled fallback* [43] overcomes these limitations by parallelizing the fallback using subgroup operations. Since the synchronization and progress guarantees of subgroup operations are not yet clear[9], Dartagnan cannot support them. Thus, we skip the decoupled fallback implementation.

9 Related Work

Most previous works on (non-)termination are either restricted to sequential programs [11, 15, 23, 33], or consider concurrency with interleaving semantics and strong scheduler assumptions [7, 30]. However, three lines of work are closely related to ours. We discuss them below.

Abdulla et al. have considered the verification of liveness properties under weak memory models [1]. They rely on a unifying operational semantics that captures several memory models (but not all, such as ARM8 and Power) and show how to reduce the repeated control reachability problem

[7]Proving safety additionally requires reasoning about weak consistency.

[8]Details of our semantics for control barriers can be found in the Appendix.

[9]This is a topic the memory and execution model TSG of the Khronos group is currently working on [19].

Table 3. Termination of prefixsum implementations under different forward progress guarantees.

PrefixSum	Scheduler	Terminates	Time
Ours (ids)	FAIR	✓	22.2s (B=3)
Ours (ids)	OBE	✗	1m 18s (B=3)
Ours (ids)	HSA	✓	22.8s (B=3)
Ours (ids)	UNFAIR	✗	1m 26s (B=3)
Ours (ticket)	FAIR	✓	19.4s (B=3)
Ours (ticket)	OBE	✓	22.9s (B=3)
Ours (ticket)	LOBE	✓	23.2s (B=3)
Ours (ticket)	UNFAIR	✗	51.1s (B=3)

PrefixSum	Scheduler	Terminates	Time
UCSC (ticket)	FAIR	✓	6.4s (B=2)
UCSC (ticket)	OBE	✓	7.8s (B=2)
UCSC (ticket)	HSA	✗	10.2s (B=2)
UCSC (ticket)	UNFAIR	✗	18.8s (B=2)
Vello (ticket)	FAIR	✓	13m 50s (B=8)
Vello (ticket)	OBE	✓	21m 51s (B=8)
Vello (ticket)	LOBE	✓	15m 20s (B=8)
Vello (ticket)	UNFAIR	✓	24m 49s (B=8)

to a state reachability problem under weak memory. This reduction makes proof techniques for reachability also applicable to liveness verification and even yields decidability results. However, the reduction relies on memory and scheduler fairness assumptions stronger than ours, in particular, they assume that any transition that can happen infinitely often will happen infinitely often. This excludes unfair scheduling such as found on GPUs, but also reasonable weak behavior where, e.g, two store operations are always propagated out-of-order or a store buffer is never emptied fully. Also, their work gives no tool to actually solve the non-termination problem.

Another important work is by Lahav et al. about memory fairness under axiomatic memory models [22]. They characterize memory-fair behavior directly on execution graphs using prefix-finiteness of certain communication edges in the graphs. This notion of memory fairness relates to a weaker notion of fairness in the operational semantics: only transitions that are continuously enabled will eventually happen. They use this characterization to prove (non-)termination of programs (under fair scheduling) whose only unbounded behavior comes from spin loops, i.e., loops that do not have side effects and only read from memory. This characterization also gives a practical algorithm to automatically check (non-)termination under weak memory [22, 34] which is implemented in GenMC. Our development makes use of Lahav et al.'s characterization of memory fairness, but also considers unfair scheduling. While our approach is restricted to non-termination only, it can deal with a far richer class of loops, in particular, loops with side effects.

The interest in unfair schedulers has developed relatively recently with the rising interest in executing complex parallel algorithms on GPUs. In [45], Sorensen et al. analyzed the semi-fair scheduling guarantees that existing GPU programs rely on and gave a formal description of these guarantees. In [46], they tested if actual GPU implementations provide the formalized scheduler guarantees. To do so, they developed a termination oracle that decides the termination of small litmus programs under semi-fair schedulers by exhaustive enumeration techniques. They disregard weak memory behaviors and work on classical operational interleaving semantics of concurrent programs. However, it is known that GPUs can exhibit weak memory behaviors [20, 26, 28].

10 Data Availability Statement

An accompanying artifact is available to reproduce all data presented in Section 8 [16]. The full Dartagnan tool is open-source and available at https://github.com/hernanponcedeleon/Dat3M.

References

[1] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Adwait Godbole, Shankaranarayanan Krishna, and Mihir Vahanwala. 2023. Overcoming Memory Weakness with Unified Fairness. In *Computer Aided Verification (CAV)*, Constantin Enea and Akash Lal (Eds.). Springer Nature Switzerland, Cham, 184–205. doi:10.1007/978-3-031-37706-8_10

[2] Andy Adinets and Duane Merrill. 2022. Onesweep: A Faster Least Significant Digit Radix Sort for GPUs. arXiv:2206.01784 [cs.DC] https://arxiv.org/abs/2206.01784

[3] Jade Alglave. 2010. *A Shared Memory Poetics.* Thèse de doctorat. L'université Paris Denis Diderot.

[4] Jade Alglave, Patrick Cousot, and Luc Maranget. 2016. Syntax and semantics of the weak consistency model specification language CAT. *CoRR* abs/1608.07531 (2016). doi:10.48550/arXiv.1608.07531

[5] Jade Alglave, Will Deacon, Richard Grisenthwaite, Antoine Hacquard, and Luc Maranget. 2021. Armed Cats: Formal Concurrency Modelling at Arm. *ACM Trans. Program. Lang. Syst.* 43, 2 (2021), 8:1–8:54. doi:10.1145/3458926

[6] Jade Alglave, Luc Maranget, and Michael Tautschnig. 2014. Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory. *ACM Trans. Program. Lang. Syst.* 36, 2 (2014), 7:1–7:74. doi:10.1145/2627752

[7] Mohamed Faouzi Atig, Ahmed Bouajjani, Michael Emmi, and Akash Lal. 2012. Detecting Fair Non-termination in Multithreaded Programs. In *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings (Lecture Notes in Computer Science, Vol. 7358)*, P. Madhusudan and Sanjit A. Seshia (Eds.). Springer, 210–226. doi:10.1007/978-3-642-31424-7_19

[8] Mark Batty, Alastair F. Donaldson, and John Wickerson. 2016. Overhauling SC atomics in C11 and OpenCL. In *POPL*. ACM, 634–648. doi:10.1145/2837614.2837637

[9] Dirk Beyer and Jan Strejcek. 2025. Improvements in Software Verification and Witness Validation: SV-COMP 2025. In *Tools and Algorithms for the Construction and Analysis of Systems - 31st International Conference, TACAS 2025, Held as Part of the International Joint Conferences on Theory and Practice of Software, ETAPS 2025, Hamilton, ON, Canada, May 3-8, 2025, Proceedings, Part III (Lecture Notes in Computer Science, Vol. 15698)*, Arie Gurfinkel and Marijn Heule (Eds.). Springer, 151–186. doi:10.1007/978-3-031-90660-2_9

[10] Guy E. Blelloch. 2018. Prefix sums and their applications. (6 2018). doi:10.1184/R1/6608579.v1

[11] Hong-Yi Chen, Byron Cook, Carsten Fuhs, Kaustubh Nimkar, and Peter O'Hearn. 2014. Proving Nontermination via Safety. In *Tools and Algorithms for the Construction and Analysis of Systems*, Erika Ábrahám and Klaus Havelund (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 156–171. doi:10.1007/978-3-642-54862-8_11

[12] Yu-Fang Chen, Matthias Heizmann, Ondrej Lengál, Yong Li, Ming-Hsien Tsai, Andrea Turrini, and Lijun Zhang. 2018. Advanced automata-based algorithms for program termination checking. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, Jeffrey S. Foster and Dan Grossman (Eds.). ACM, 135–150. doi:10.1145/3192366.3192405

[13] James V. Contini. 2025. *Scanbox: A tunable and portable GPU prefix scan implementation in Vulkan and WebGPU.* Bachelor's thesis. University of California Santa Cruz.

[14] Nissim Francez. 1986. *Fairness.* Springer. doi:10.1007/978-1-4612-4886-6

[15] Ashutosh Gupta, Thomas A. Henzinger, Rupak Majumdar, Andrey Rybalchenko, and Ru-Gang Xu. 2008. Proving non-termination. *SIGPLAN Not.* 43, 1 (Jan. 2008), 147–158. doi:10.1145/1328897.1328459

[16] Thomas Haas, Roland Meyer, Ponce de Leon Hernan, and Andrés Lomeli. 2025. Recurrence Sets for Proving Fair Non-termination under Axiomatic Memory Consistency Models (Artifact). doi:10.5281/zenodo.17770530

[17] Thomas Haas, Roland Meyer, and Hernán Ponce de León. 2022. CAAT: Consistency as a Theory. *Proc. ACM Program. Lang.* 6, OOPSLA2 (2022). doi:10.1145/3563292

[18] Alastair Harrison. 2024. [libc++] Incorrect memory order in atomic wait. https://github.com/llvm/llvm-project/issues/109290

[19] Mateusz Kielan. 2025. Do Subgroup Shuffle Intrinsics require a preceeding OpControlBarrier with scope at least Subgroup to avoid UB? https://github.com/KhronosGroup/Vulkan-Docs/issues/2548

[20] Jake Kirkham, Tyler Sorensen, Esin Tureci, and Margaret Martonosi. 2020. Foundations of empirical memory consistency testing. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 226:1–226:29. doi:10.1145/3428294

[21] Michalis Kokologiannakis and Viktor Vafeiadis. 2021. GenMC: A Model Checker for Weak Memory Models. In *CAV (LNCS, Vol. 12759)*. Springer, 427–440. doi:10.1007/978-3-030-81685-8_20

[22] Ori Lahav, Egor Namakonov, Jonas Oberhauser, Anton Podkopaev, and Viktor Vafeiadis. 2021. Making weak memory models fair. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–27. doi:10.1145/3485475

[23] Jan Leike and Matthias Heizmann. 2018. Geometric Nontermination Arguments. In *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 10806)*, Dirk Beyer and Marieke Huisman (Eds.). Springer, 266–283. doi:10.1007/978-3-319-89963-3_16

[24] Raph Levien. 2020. Prefix sum on Vulkan. https://raphlinus.github.io/gpu/2020/04/30/prefix-sum.html.

[25] Raph Levien. 2021. An HLSL implementation of prefix-scan. https://github.com/linebender/vello/blob/custom-hal-archive-with-shaders/tests/shader/prefix.comp.

[26] Reese Levine, Mingun Cho, Devon McKee, Andrew Quinn, and Tyler Sorensen. 2023. GPUHarbor: Testing GPU Memory Consistency at Large (Experience Paper). In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*, René Just and Gordon Fraser (Eds.). ACM, 779–791. doi:10.1145/3597926.3598095

[27] Reese Levine and James Contini. 2023. An OpenCL implementation of prefix-scan. https://github.com/reeselevine/vk-prefix-scan.

[28] Reese Levine, Tianhao Guo, Mingun Cho, Alan Baker, Raph Levien, David Neto, Andrew Quinn, and Tyler Sorensen. 2023. MC Mutants: Evaluating and Improving Testing for Memory Consistency Specifications. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, Tor M. Aamodt, Natalie D. Enright Jerger, and Michael M. Swift (Eds.). ACM, 473–488. doi:10.1145/3575693.3575750

[29] Daniel Lustig, Sameer Sahasrabuddhe, and Olivier Giroux. 2019. A Formal Analysis of the NVIDIA PTX Memory Consistency Model. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, Iris Bahar, Maurice Herlihy, Emmett Witchel, and Alvin R. Lebeck (Eds.). ACM, 257–270. doi:10.1145/3297858.3304043

[30] Zohar Manna and Amir Pnueli. 1992. *The temporal logic of reactive and concurrent systems.* Springer-Verlag, Berlin, Heidelberg. doi:10.1007/978-1-4612-0931-7

[31] Jeremy Manson, William Pugh, and Sarita V. Adve. 2006. The Java memory model. In *POPL*. ACM, 378–391. doi:10.1145/1047659.1040336

[32] Duane Merrill and Michael Garland. 2016. Single-pass Parallel Prefix Scan with Decoupled Look-back. https://research.nvidia.com/publication/2016-03_single-pass-parallel-prefix-scan-decoupled-look-back.

[33] Ravindra Metta, Hrishikesh Karmarkar, Kumar Madhukar, R. Venkatesh, and Supratik Chakraborty. 2024. PROTON: PRObes for Termination Or Not (Competition Contribution). In *Tools and Algorithms for the Construction and Analysis of Systems - 30th International Conference, TACAS 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings, Part III (Lecture Notes in Computer Science, Vol. 14572)*, Bernd Finkbeiner and Laura Kovács (Eds.). Springer, 393–398. doi:10.1007/978-3-031-57256-2_27

[34] Jonas Oberhauser, Rafael Lourenco de Lima Chehab, Diogo Behrens, Ming Fu, Antonio Paolillo, Lilith Oberhauser, Koustubha Bhat, Yuzhong Wen, Haibo Chen, Jaeho Kim, and Viktor Vafeiadis. 2021. VSync: push-button verification and optimization for synchronization primitives on weak memory models. In *ASPLOS*. ACM, 530–545. doi:10.1145/3445814.3446748

[35] Anton Podkopaev, Ori Lahav, and Viktor Vafeiadis. 2019. Bridging the gap between programming languages and hardware weak memory models. *PACMPL* 3, POPL (2019), 69:1–69:31. doi:10.1145/3290382

[36] Hernán Ponce de León, Florian Furbach, Keijo Heljanko, and Roland Meyer. 2017. Portability Analysis for Weak Memory Models. PORTHOS: One Tool for all Models. In *SAS (LNCS, Vol. 10422)*. Springer, 299–320. doi:10.1007/978-3-319-66706-5_15

[37] Hernán Ponce de León, Florian Furbach, Keijo Heljanko, and Roland Meyer. 2018. BMC with Memory Models as Modules. In *FMCAD*. IEEE, 1–9. doi:10.23919/FMCAD.2018.8603021

[38] Hernán Ponce de León, Florian Furbach, Keijo Heljanko, and Roland Meyer. 2020. Dartagnan: Bounded Model Checking for Weak Memory Models (Competition Contribution). In *TACAS (2) (LNCS, Vol. 12079)*. Springer, 378–382. doi:10.1007/978-3-030-45237-7_24

[39] S4C Safe and Scalable System Software Concurrency. 2024. libvsync. https://github.com/open-s4c/libvsync.

[40] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. 2011. Understanding POWER multiprocessors. In *PLDI*. ACM, 175–186. doi:10.1145/1993498.1993520

[41] Frank Schüssele. 2022. New subcategory: Termination concurrent. https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks/-/merge_requests/1363

[42] Shubhabrata Sengupta, Mark J. Harris, Yao Zhang, and John D. Owens. 2007. Scan primitives for GPU computing. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware 2007, San Diego, California, USA, August 4-5, 2007*, Mark Segal and Timo Aila (Eds.). Eurographics Association, 97–106. doi:10.2312/EGGH/EGGH07/097-106

[43] Thomas Smith, Raph Levien, and John D. Owens. 2025. Decoupled Fallback: A Portable Single-Pass GPU Scan. In *Proceedings of the 37th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '25)*. doi:10.1145/3694906.3743326

[44] Tyler Sorensen, Alastair F. Donaldson, Mark Batty, Ganesh Gopalakrishnan, and Zvonimir Rakamaric. 2016. Portable inter-workgroup barrier synchronisation for GPUs. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, Eelco Visser and Yannis Smaragdakis (Eds.). ACM, 39–58. doi:10.1145/2983990.2984032

[45] Tyler Sorensen, Hugues Evrard, and Alastair F. Donaldson. 2018. GPU Schedulers: How Fair Is Fair Enough?. In *29th International Conference on Concurrency Theory (CONCUR 2018) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 118)*, Sven Schewe and Lijun Zhang (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 23:1–23:17. doi:10.4230/LIPIcs.CONCUR.2018.23

[46] Tyler Sorensen, Lucas F. Salvador, Harmit Raval, Hugues Evrard, John Wickerson, Margaret Martonosi, and Alastair F. Donaldson. 2021. Specifying and testing GPU workgroup progress models. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–30. doi:10.1145/3485508

[47] Haining Tong, Natalia Gavrilenko, Hernán Ponce de León, and Keijo Heljanko. 2024. Towards Unified Analysis of GPU Consistency. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4, ASPLOS 2024, Hilton La Jolla Torrey Pines, La Jolla, CA, USA, 27 April 2024 - 1 May 2024*, Rajiv Gupta, Nael B. Abu-Ghazaleh, Madan Musuvathi, and Dan Tsafrir (Eds.). ACM, 329–344. doi:10.1145/3622781.3674174

Table 4. Termination of prefixsum implementations under different forward progress guarantees.

PrefixSum	Scheduler	Terminates	Time
Ours (ids)	FAIR	✓	22.2s (B=3)
Ours (ids)	OBE	✗	1m 18s (B=3)
Ours (ids)	HSA	✓	22.8s (B=3)
Ours (ids)	HSA_OBE	✓	22.9s (B=3)
Ours (ids)	LOBE	✓	24.4s (B=3)
Ours (ids)	UNFAIR	✗	1m 26s (B=3)
Ours (ticket)	FAIR	✓	19.4s (B=3)
Ours (ticket)	OBE	✓	22.9s (B=3)
Ours (ticket)	HSA	✗	55.4s (B=3)
Ours (ticket)	HSA_OBE	✓	21.3s (B=3)
Ours (ticket)	LOBE	✓	23.2s (B=3)
Ours (ticket)	UNFAIR	✗	51.1s (B=3)
UCSC (ticket)	FAIR	✓	6.4s (B=2)
UCSC (ticket)	OBE	✓	7.8s (B=2)
UCSC (ticket)	HSA	✗	10.2s (B=2)
UCSC (ticket)	HSA_OBE	✓	7.4s (B=2)
UCSC (ticket)	LOBE	✓	7.5s (B=2)
UCSC (ticket)	UNFAIR	✗	18.8s (B=2)
Vello (ticket)	FAIR	✓	13m 50s (B=8)
Vello (ticket)	OBE	✓	21m 51s (B=8)
Vello (ticket)	HSA	✓	18m 39s (B=8)
Vello (ticket)	HSA_OBE	✓	18m 37s (B=8)
Vello (ticket)	LOBE	✓	15m 20s (B=8)
Vello (ticket)	UNFAIR	✓	24m 49s (B=8)

A Full Evaluation

The full set of experiments from Table 3 is given in Table 4.

B Proofs

Proof of Theorem 6.6. Let $\xi \sqsubseteq \xi'$ be an extension in the abstract. The run of the extended abstract execution prefix has shape $\xi'.\rho = \rho(\xi).\rho_{new}$ and the newly added events are given by $X_{new} = \mathrm{Ev}(\rho_{new})$. Now, consider a concrete execution prefix $\varepsilon \in \gamma(\xi)$ which has a run of shape $\varepsilon.\rho = \rho_{pre}.\rho(\xi)$. We can extend this run to $\rho' = \rho_{pre}.\rho(\xi).\rho_{new} = \rho_{pre}.\rho(\xi')$. We construct XG' by starting from $EG(\rho')$ and adding read-from and coherence edges as follows. First, we add all read-from and coherence edges that already exist in $\varepsilon.XG$ and $\xi'.XG$. Then, for each prefix-justified read $x = \mathrm{R}(l, \xi.\mu(l)) \in X_{new}$, we identify the co-maximal write y to address l among all writes in $X_{pre} = \mathrm{Ev}(\rho_{pre})$. This write must exist and its value must be $\xi.\mu(l)$ by definition of the concretization. We can add the $\mathrm{rf}(y, x)$-edge to justify the read.

For each write event in X_{new} we place it co-after all same-address writes in X_{pre}. This yields the graph XG'. For this to be a valid execution graph, we need to check that its coherence is total per address, i.e., it relates all same-address writes and is acyclic. To see this, first notice that all writes in $XG'.X \setminus X_{new} = \varepsilon.X$ are totally ordered as desired. Similarly, the writes in $XG'.X \setminus X_{pre} = \xi'.X$ are totally ordered. Observe that the two suborders are compatible: their union is acyclic. If it was not, there would be a co-cycle involving events from X_{pre} and X_{new}, in particular, there would be a co-path from X_{new} to X_{pre}. By definition of the extension $\xi \sqsubseteq \xi'$ this cannot be the case, because we do not permit (abstract) $\mathrm{co}(X_{new}, \bullet)$-edges. We can now make the order total by adding co-edges from X_{pre} to X_{new}. This yields the finished execution graph XG'.

Lastly, we define $\pi' = \varepsilon.\pi \cup \xi'.\pi$ Putting all together, we get $\varepsilon' = (\rho', XG', \pi')$ which satisfies $\varepsilon \sqsubseteq \varepsilon'$. Furthermore, it is easy to see that by construction that $\varepsilon' \in \gamma(\xi')$ as desired. □

Proof of Theorem 6.7. Let $\xi \sqsubseteq \xi'$ and let G be the combined and partially unfolded execution graph. Let $S = \xi'.X \setminus \xi.X$ be the newly appended events and let mm' the instrumented memory relative to S.

First we show that if G is consistent with mm, then $\xi \sqsubseteq \xi'$ is consistent. Let $\varepsilon \in \gamma(\xi)$ be a consistent execution and let $\varepsilon \sqsubseteq \varepsilon' \in \gamma(\xi')$ be its canonical extension. We need to show that ε' is consistent. Suppose ξ' was inconsistent w.r.t. some axiom **empty**(r) in mm (wlog., we restrict to emptiness axioms because all axioms are reducible to those). Then there exists a $\mathsf{r}(x, y)$-edge in ε'. We consider the derivation tree of $\mathsf{r}(x, y)$ induced by the derivations that mm performs to compute the edge. The leaves of this derivation tree are edges of base relations. Among those edges, there must be some that involves events from S, for otherwise, that derivation is also valid on $\varepsilon.XG$ (the memory model is monotonic) which contradicts the consistency of ε. By construction of the instrumented model mm', this derivation tree is essentially also a derivation tree for $\mathsf{r}'(x, y)$ in $\varepsilon'.XG$. Furthermore, the edge remains derivable when we collapse events and edges, and therefore, it is also derivable in G. Hence, G is also inconsistent with mm', which contradicts the assumption. It follows that ε' must be consistent as desired.

The second part of the lemma states that if G has no $\mathsf{r}^*; \mathsf{r}'; \mathsf{r}^*$-edges into any event of $\xi.\pi(\mathsf{r}) \cup \{\bullet\}$ then $\xi \sqsubseteq \xi'$ is r-memory-fair. The proof is similar to above. Let $\varepsilon \in \gamma(\xi)$ be an execution and let $\varepsilon \sqsubseteq \varepsilon' \in \gamma(\xi')$ be its canonical extension. We need to show that $\varepsilon \sqsubseteq \varepsilon'$ is r-memory-fair. Suppose it was not, then there is an $x \in \varepsilon.\pi(\mathsf{r})$ whose prefix got extended, i.e., a new edge $\mathsf{r}^+(y, x)$ is derivable in $\varepsilon'.XG$. We again consider the derivation tree of that edge and notice that it must involve some newly appended events S (for otherwise, the edge would not be new). This means the derivation tree also derives a $\mathsf{r}^*; \mathsf{r}'; \mathsf{r}^*(y, x)$-edge in $\varepsilon'.X$. Since collapsing preserves edges, this edge must be represented in G. Now observe that since x is r-prefix-complete, it gets collapsed to an event inside $\xi.\pi(\mathsf{r}) \cup \{\bullet\}$, which yields a contradiction with the fact that G has no such edges into that set of events. □

C Notes on continuity

In the main text, we shortly defined lower semi-continuity and showed how it plays an important role in the soundness of recurrence sets. Here, we give a more complete picture about continuity, including definitions of upper semi-continuity and (full) continuity, and relevant lemmas on their composition properties. This will justify Claim 1 and, in particular, why the alternation between negations and projective operators as discussed in Section 5.2 is necessary to construct non-lower-semi-continuous memory models.

Definition C.1 (Set-theoretic limits). Let $(A_i)_{i \in \mathbb{N}}$ be a sequence of sets. Then the limit inferior $\liminf_i A_i := \bigcup_i \bigcap_{j \geq i} A_i$ and the limit superior $\limsup_i A_i := \bigcap_i \bigcup_{j \geq i} A_i$ both exists. Notice that $\liminf A_i \leq \limsup A_i$ always holds. If both coincide, we can define the limit $\lim_i A_i := \liminf_i A_i = \limsup_i A_i$.

Definition C.2 (Semi-continuous functions). A function $f : \mathbb{P}(\mathbf{A}) \to \mathbb{P}(\mathbb{B})$ is lower semi-continuous if for all sequences $(A_i)_{i \in \mathbb{N}}$ we have $\liminf f(A_i) \geq f(\liminf a_i)$. Similarly, f is upper semi-continuous if $\limsup f(A_i) \leq f(\limsup A_i)$. If a function is both lower and upper semi-continuous, we call it continuous.

Lemma C.3. *Continuous functions preserve limits.*

Proof. Let f be continuous and A_i be a sequence with limit A. Then $f(A) = f(\liminf_i A_i) \leq \liminf_i f(A_i) \leq \limsup_i f(A_i) \leq f(\limsup_i A_i) = f(A)$. □

Lemma C.4. *If f and g are continuous functions that compose, then $f \circ g$ is also continuous.*

Proof. Let A_i be a sequence with limit A, i.e., $\lim_i A_i = \liminf_i A_i = \limsup_i A_i = A$. Then $\lim_i f(g(A_i)) = f(\lim_i g(A_i)))f(g(\lim_i A_i)) = f(g(A))$ by applying continuity of f and g. □

Lemma C.5. *If f and g are lower (upper) semi-continuous and f is monotonic, then the composition $f \circ g$ is lower (upper) semi-continuous.*

Lemma C.6. *Let A_i be a sequence. We consider the case where f and g are lower semi-continuous. We have $\liminf_i f(g(A_i)) \geq f(\liminf_i g(A_i)) \geq f(g(\liminf_i A_i))$. The first inequality holds by lower semi-cont. of f and the second holds by lower semi-cont. of g and the fact that f is monotonic. In the case where f and g are upper semi-continuous, we have $\limsup_i f(g(A_i)) \leq f(\limsup_i g(A_i)) \leq f(g(\limsup_i A_i))$, by essentially the same arguments as above.*

The definitions of continuity can be applied to functions f from ω-complete partial orders $\mathbb{A}$ into power sets, by only considering monotonic sequences $(a_i)_{i\in\mathbb{N}}$ and defining $\lim_i a_i = \liminf_i a_i = \limsup_i a_i$. Notice how we derive limit inferior and superior from the existence of unique limits of ω-chains, opposite to how limits on sets are defined by limit inferior and superior. The restriction to ω-chains gives us stronger results.

Lemma C.7. *If $f : \mathbb{A} \to \mathbb{P}(\mathbb{B})$ is monotonic and lower semi-continuous, then it is also continuous.*

Proof. We need to show that f is upper semi-continuous. To see this, consider an ω-chain $(a_i)_{i\in\mathbb{N}}$ with limit a, then we have $\limsup_i f(a_i) \leq \limsup_i f(a) = f(a) = f(\limsup a_i)$. The first inequality holds by monotonicity and the fact that $a_i \leq a$ for all i, and the last equality holds by $a = \lim_i a_i = \limsup_i a_i$. □

Lemma C.8. *If $f : \mathbb{P}(\mathbb{A}) \to \mathbb{P}(\mathbb{B})$ is Scott-continuous, then it is also lower semi-continuous. This also holds true if $\mathbb{P}(\mathbb{A})$ is replaced by a ω-complete partial order.*

Proof. We have $\liminf_i f(a_i) = \bigcup_j \bigcap_{i\geq j} f(a_i) \geq \bigcup_j f(\bigcap_{i\geq j} a_i) = f(\bigcup_j \bigcap_{i\geq j} a_i) = f(\liminf_i a_i)$. The first inequality holds by monotonicity of f (Scott-continuous functions are monotonic) and the fact that $a_k \geq \bigcap_{i\geq k} a_i$ for all i. The following equality holds by Scott-continuity of f which guarantees preservation of limits of increasing sequences. Notice that $\bigcap_{j\geq i} a_i$ is increasing w.r.t. j, because the larger j, the less we intersect. □

Combining the previous two lemmas, we get the following corollary.

Corollary C.9. *If $f : \mathbb{A} \to \mathbb{P}(\mathbb{B})$ is Scott-continuous, then it is also continuous.*

Lemma C.10 (Continuity of set operations). *The set operations union and intersection are monotonic and continuous. Set complementation is continuous but not monotonic (but antitonic). The cartesian product operation is monotonic and lower semi-continuous, but not upper semi-continuous. The projective operations (composition, range projection, and domain projection) are also monotonic and lower semi-continuous. The converse operation is continuous.*

Actually, all operations but complementation are also Scott-continuous.

Notice that the above lemma talks about set operations as functions from power sets into power sets. This makes a subtle difference in properties. For example, compare the composition function $(\mathsf{r}_1, \mathsf{r}_2) \mapsto \mathsf{r}_1; \mathsf{r}_2$ vs. the function $\varepsilon \mapsto \varepsilon.\mathsf{r}_1; \varepsilon.\mathsf{r}_2$. The former is not upper semi-continuous and hence not continuous, but the latter is! This comes from the fact that the latter only needs to reason about monotonic input sequences rather than arbitrary input sequences. This difference makes it particularly hard to construct, in CAT, derived relations that fail to be lower semi-continuous and is the reason why Claim 1 holds. Let us demonstrate this.

Our goal is to construct a relation r that is not lower semi-continuous when understood as a function $\mathsf{r} : (\mathsf{ExecPre}, \sqsubseteq) \to \mathbb{P}(X \times X)$. Since all operators in the CAT language except for negation/difference are monotonic and lower semi-continuous and composing those operators again yields monotonic and lower semi-continuous operators, we need to use negations to construct the desired r. However, since the negation operator is continuous, and on base relations and any boolean combination thereof it is even monotonic (by a property we explained in the main text), we need to apply the negation after a projective operator. So let r_1 be a relation defined by using at least one projective operator but no negations yet. r_1 is monotonic and lower semi-continuous and hence even continuous by a previous lemma. We apply the negation to get $\mathsf{r}_2 = \neg\mathsf{r}_1$ which is in general non-monotonic but still continuous, since composition of continuous functions is again continuous. Applying more continuous operators like the boolean set operations will get us nowhere. We now need to apply yet again a projective operator, which leads to a relation r_3 that fails to be continuous but is still lower semi-continuous. Lower semi-continuity still holds by a previous lemma about the composition of a monotonic lower semi-continuous function (the projective operator) and another semi-continuous function (r_2). Notice that r_3 can only fail to be continuous here because it is not monotonic (that is why we had to construct r_2 to be not monotonic). Now composing it further with any of the positive operators will keep the relation lower semi-continuous, so we need to apply yet another negation to finally get the desired relation $\mathsf{r} = \mathsf{r}_4$ that fails to be lower semi-continuous. In summary, we need to perform at least the following sequence of operations "projection -> negation -> projection -> negation" to define a relation in CAT that fails to be lower semi-continuous.

D Extension to Control Barriers

GPU code uses control barriers to synchronize the control-flow of multiple threads. On entering a control barrier, a thread blocks until all other threads within the same execution scope, e.g., the same workgroup, reach that control barrier. To model this operation axiomatically, we add a new control barrier event $CB(id, T, status)$ that consists of an id, a set of threads $T \subseteq \mathbf{Tid}$, and a status of whether the thread got blocked or not. The intuitive semantics of executing a control barrier is that the executing thread blocks until all threads in T execute their corresponding control barrier with the same id (the id is used to identify which control barriers are related to each other). We assume that the id and the set of threads T synchronizing at a barrier is statically determined, which is the case for so-called uniform control barriers. Upon executing a control barrier, a thread guesses whether it gets blocked or whether it will pass the barrier. In case of the former, we assumes the thread records its blocking state, including the set of threads T it is waiting for, in its local state.

In a program execution, these guesses need to get justified, similar to how guessed read values get justified. For this, we use a new sync relation that tries to relate the k-th occurrence of an event $x = CB(id, T, status)$ of a thread to the k-th occurrence of $y = CB(id, T, status')$ in all other threads in T. If the relation sync can be established, then the status of all matching control barriers must be "unblocked", otherwise, they are "blocked". Partial execution graphs are allowed to have unblocked, but yet unjustified control barriers. Recurrence sets then have the extra condition of eventually justifying every unblocked control barrier (similar to how they eventually justify every read). This ensures that the infinite execution witnessed by the recurrence set is feasible in that it does not eventually get stuck in a control barrier. At the same time, the scheduling condition gets relaxed so that only fairly scheduled threads that are unblocked need to make progress. For abstract recurrence sets, we add the restriction that unblocked but yet-unjustified control barriers cannot be abstracted away, i.e., they must get justified first.

Illegal unblocking behavior is prevented by the fact that unblocking must get justified. To also prevent illegal blocking behavior, we use the local state of the concrete infix: if a thread is blocked on a set of threads T and all threads in T are blocked on the same control barrier id, then the execution is invalid (they should have been unblocked).

In the presence of control barriers, there is a new kind of finite non-termination where every thread is blocked, which is witnessed by finite executions. This particular non-termination is detectable using classical reachability checks and therefore orthogonal to our recurrence sets which witness infinite non-termination.

E Improving Lasso Finding and Further Applications

We make a few remarks about how to generalize lassos further for practical applications. The presented lasso-finding approach relies on a strict repetition of the infix, meaning the values of all events must repeat exactly. The only reason for this strict requirement is that we can guarantee the control flow remains repeatable. This leads to a simple generalization where we relax the condition on dead variables and events, i.e., values of dead variables are allowed to change and so do values of unobserved stores and unused loads. More generally, we may also allow the modification of values of non-control-flow variables.

We can take it even a step further and allow variables to change as long as the the control flow stays the same. However, this requires invariant reasoning and so falls outside the scope of bounded reachability checkers such as Dartagnan. A possible solution is to look for lasso candidates that ignore the value constraints and from that candidate generate a sequential program that models the repeating infix as a single loop. The lasso candidate resolves all concurrency non-determinism in a consistent and fair manner, and hence allows us to reduce the remaining data flow to a sequential program. Then (non-)termination checkers can be used to check the sequential program [11, 15]. In fact, it suffices now to find a classical state-based recurrence set. We remark that the above idea of combining lasso-finding with invariant generation already existed in earlier works [15]. What is new is that our lassos have a more general form that considers concurrency, weak memory consistency, and fairness constraints.

Received 2025-07-10; accepted 2025-11-06