SPORE: Combining Symmetry and Partial Order Reduction

MICHALIS KOKOLOGIANNAKIS, MPI-SWS, Germany
IASON MARMANIS, MPI-SWS, Germany
VIKTOR VAFEIADIS, MPI-SWS, Germany

Symmetry reduction (SR) and partial order reduction (POR) aim to scale up model checking by exploiting the underlying program structure: SR avoids exploring executions equivalent up to some permutation of symmetric threads, while POR avoids exploring executions equivalent up to reordering of independent instructions. While both SR and POR have been well studied individually, their combination in the context of stateless model checking has remained an open problem.

In this paper, we present SPORE, the first stateless model checker that combines SR and POR in a sound, complete and optimal manner. SPORE can leverage both symmetries in the client program itself, but also *internal symmetries* in the underlying implementation (i.e., idempotent operations), a novel symmetry notion we introduce in this paper. Our experiments confirm that SPORE explores drastically fewer executions than tools that solely employ SR/POR, thereby greatly advancing the state-of-the-art.

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

Additional Key Words and Phrases: Model Checking, Dynamic Partial Order Reduction, Symmetry Reduction

ACM Reference Format:
Michalis Kokologiannakis, Iason Marmanis, and Viktor Vafeiadis. 2024. SPORE: Combining Symmetry and Partial Order Reduction. *Proc. ACM Program. Lang.* 8, PLDI, Article 219 (June 2024), 58 pages. https://doi.org/10.1145/3656449

1 Introduction

Stateless model checking (SMC) [Godefroid 1997] verifies a concurrent program by enumerating all of its executions. SMC is quite popular in concurrent program verification as (a) can be used by programmers without any expertise in formal methods, (b) it can handle programs in full-fledged programming languages like C, C++ and Java, and (c) it can reason about the effects of the underlying weak memory model (e.g., C/C++11 [Lahav et al. 2017]). On the downside, however, SMC only supports verification of bounded programs, and often does not scale well enough to handle client programs with a sufficient number of threads to provide strong confidence the correctness of a given implementation.

There are two sound techniques that can be employed to increase the scalability of SMC.

Symmetry reduction (SR) [Clarke et al. 1996; Emerson and Wahl 2005] exploits symmetries in the threads of the program under test (e.g., all threads running the same code) and avoids to consider all the ways in which symmetric threads interleave, as the order in which such threads execute is

Authors' Contact Information: Michalis Kokologiannakis, MPI-SWS, Kaiserslautern, Germany, michalis@mpi-sws.org; Iason Marmanis, MPI-SWS, Kaiserslautern, Germany, imarmanis@mpi-sws.org; Viktor Vafeiadis, MPI-SWS, Kaiserslautern, Germany, viktor@mpi-sws.org.

ACM 2475-1421/2024/6-ART219
https://doi.org/10.1145/3656449

clearly irrelevant. As an example of SR, consider the FAIS program where N symmetric threads perform an atomic "fetch-and-increment" operation on x:

$$\texttt{fetch\_add}(x, 1) \;\Big\|\; \dots \;\Big\|\; \texttt{fetch\_add}(x, 1) \qquad \text{(FAIS)}$$

While naive SMC explores $N!$ executions for this program, SR only explores 1 execution.

Dynamic partial order reduction (DPOR) [Abdulla et al. 2014; Flanagan and Godefroid 2005] reduces the program state space by not exploring executions that are equivalent up to some permutation of independent instructions (e.g., instructions accessing different variables). For instance, consider the program below where 26 (non-symmetric) threads write different parts of an array:

$$a := 1 \;\Big\|\; b := 2 \;\Big\|\; \dots \;\Big\|\; z := 26 \qquad \text{(ARRAY)}$$

For ARRAY, naive SMC would again explore 26! executions while DPOR would only explore 1, as it notices that all threads access different parts of memory, and hence their relative order is irrelevant.

A common way to view both SR and DPOR is via the *equivalence partitioning* they induce on the program state space. Indeed, SR groups together executions that can be obtained from one another by changing the ID of symmetric threads, while DPOR groups together executions that can be obtained from one another by changing the order of non-conflicting instructions.

Observe, however, that even for symmetric programs, SR and DPOR are not equivalent, and neither approach subsumes the other. This can be seen with the example below:

$$\begin{array}{l} i := \texttt{fetch\_add}(x, 1) \\ a[i] := i \end{array} \;\Bigg\|\; \dots \;\Bigg\|\; \begin{array}{l} i := \texttt{fetch\_add}(x, 1) \\ a[i] := i \end{array} \qquad \text{(FAIS+ARRAY)}$$

While DPOR explores $N!$ executions for FAIS+ARRAY (due to the conflicting `fetch_add`s), SR explores $(2N-1)!!$ executions (double factorial of odd numbers). This discrepancy is because in SMC, after each thread has executed its `fetch_add`, symmetry "breaks", as each thread reads a different value.

Even though SR and DPOR are both effective when applied, existing SR/DROR approaches have two major limitations. First, they are incompatible: indeed, despite years of research on each of SR/DPOR, no algorithm manages to successfully combine the two, so employing one of them precludes the usage of the other. Second, both SR and DPOR fail to leverage *internal symmetries*, i.e., *idempotent* operations of the underlying implementation. One case of internal symmetry is the quintessential *helping* pattern, where some operation observes an ongoing operations of the same type that is incomplete, and then tries to complete the ongoing operation before performing its own. SR fails to exploit internal symmetries as the threads performing the operations are not sharing the same code, while DPOR fails to do so because the two operations are considered conflicting.

In this paper, we present SPORE (Symmetry and Partial Order Reduction Explorer), a novel algorithm that combines SR and DPOR, and overcomes both limitations above. SPORE resolves thread-level symmetries by restricting the coherence order of symmetric conflicting operations to agree with their thread order, and internal symmetries with a novel memory-model axiomatization that equates executions differing only in the order of the locally symmetric operations. The resulting algorithm is sound, complete and optimal under the combined equivalence partitionings, and achieves exponential reduction in verification time over the state-of-the-art. SPORE is also parametric in the choice of the underlying (weak) memory model.

Our contributions can be summarized as follows.

§2 We (informally) describe why the combination of DPOR and SR is non-trivial, as well as how SPORE exploits thread-level and internal internal symmetries.
§3 We present SPORE in detail and prove its correctness.
§4 We implement SPORE in a tool for C/C++ programs, and empirically demonstrate that it is orders of magnitude faster than the state-of-the-art.

2 Spore: Informal Description

We develop Spore by adding SR on top of a DPOR algorithm (as opposed to the other way around), since DPOR underpins most modern SMC solutions [Abdulla et al. 2018; Aronis et al. 2018; Chalupa et al. 2017; Kokologiannakis et al. 2022, 2019b; Norris and Demsky 2013]. As such, we begin this section by explaining the basics of DPOR (§ 2.1), and then describe why the combination of DPOR and symmetry reduction is non-trivial and how Spore achieves it (§ 2.2). We end the section by demonstrating how Spore handles internal symmetries (§ 2.3).

2.1 Dynamic Partial Order Reduction

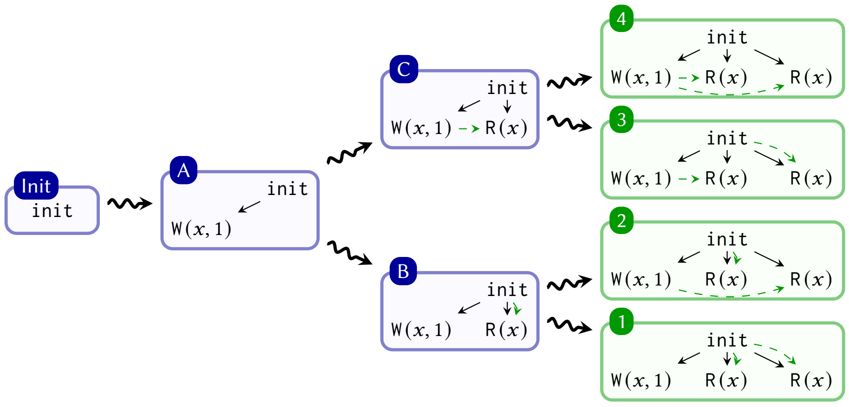
Modern DPOR algorithms, such as TruSt [Kokologiannakis et al. 2022], represent program executions up to the reordering of independent accesses in a structure called *execution graph* [Alglave et al. 2014], and verify a given program by constructing its associated execution graphs in an incremental fashion.

Each execution graph G comprises: (a) a set of events E (graph nodes), modeling instructions of the program, and (b) a few relations on events (graph edges), modeling various interactions between the instructions. In the following, we consider three such directed edges: the *program order* (po), which orders instructions of the same thread, the *reads-from* relation (rf), which relates each read event r in G to a write event w in G, from which r obtains its value, and the *coherence order* (co), which totally orders writes at each memory location.

Example 1 Consider the W+R+R program below.

$$\text{T1: } x := 1 \;\Big\|\; \text{T2: } r_2 := x \;\Big\|\; \text{T3: } r_3 := x \qquad (\text{W+R+R})$$

Under sequential consistency (SC) [Lamport 1979], the program has four executions, 1–4, which model the four equivalence classes into which the 3! = 6 thread interleavings are partitioned. These graphs can be produced by the following DPOR exploration starting from the initial graph Init through the intermediate graphs A, B, and C.



The exploration proceeds in a depth-first manner: DPOR adds the events of the program from left to right, one by one, and whenever a read has more than one place to read from, DPOR initiates a recursive subexploration. For instance, when the read of T2 is added, it can read both 0 and 1 (both options are consistent according to SC), and thus DPOR initiates subexplorations B and C. DPOR proceeds in a similar manner, until all events of the program have been added to the graph.

Conventions

Following standard conventions in the weak memory model literature, we (1) treat `rf` as a relation from the write to the read event; (2) assume a special initialization event `init`, which initializes every location with 0 and is thus po-before all other events and `co`-before all other write events; (3) we do not draw `co` edges from `init` to other writes (as it is trivially `co`-before them). In explorations, we use letters to refer to intermediate executions, numbers to refer to full executions, and red to denote executions that will not be explored.

Revisits. The exploration in Example 1 was largely straightforward, but there is still one aspect of DPOR we have not discussed: *revisiting*. For exposition purposes, suppose we add the events of W+R+R from right to left. When we encounter the reads, they cannot yet read 1 because the corresponding write does not exist in the graph. Therefore, whenever a write is added to a graph, DPOR also revisits existing same-location reads to see if they can read from the newly added write.

Whenever DPOR revisits a read r from a write w, it *restricts* the graph to remove some of the events added to the graph after r, since they may depend on the value read by r. (If not, they will be re-added in subsequent steps of the exploration.) The most common choice for restricting the graph is to keep only the events that were added before r and those causally before w (i.e., in its porf $\triangleq (\text{po} \cup \text{rf})^+$ prefix). For instance, in the right-to-left exploration of W+R+R, if $\mathtt{W}(x, 1)$ revisits the read of T3, the resulting graph does not have the read of T2 because it was added after T3 and is not porf-before $\mathtt{W}(x, 1)$.

The restriction due to revisits may lead to duplicate explorations, as we demonstrate below.

Example 2 Consider the following variation of W+R+R.

$$\text{T1: } x := 1 \;\Big\|\; \text{T2: } r_2 := x \;\Big\|\; \text{T3: } x := 2 \qquad \text{(W+R+W)}$$

Adding the events from left to right, observe that there are two subexplorations where $\mathtt{W}(x, 2)$ has the chance to revisit the read of T2: when the latter reads 0 and when it reads 1. These subexplorations are shown in Fig. 1. If $\mathtt{W}(x, 2)$ performs the revisit in both, the exact same graph will be created.

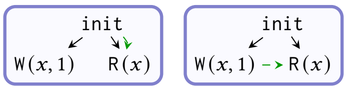


Fig. 1. Revisit opportunities

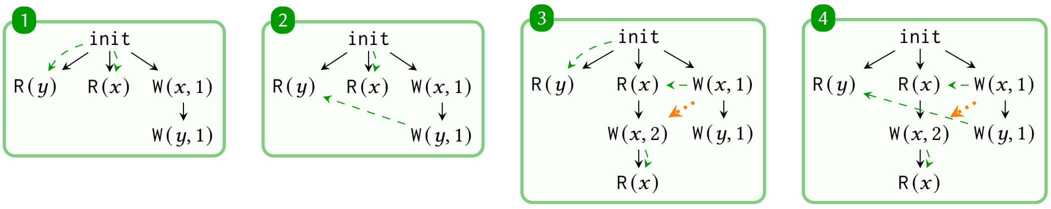
There are two ways DPOR can avoid such duplication. Abdulla et al. [2014] and Kokologiannakis et al. [2019b] simply save all encountered executions (more precisely: the ones created by revisits), and drop subsequent revisits that yield an already encountered execution. Storing executions, however, leads to exponential memory consumption in the size of the program under test.

Avoiding Duplication with Maximal Extensions. A better solution adopted by TruSt [Kokologiannakis et al. 2022] is to impose a *revisiting condition* so that a given revisit only takes place once among all possible subexplorations. The key observation is that whenever DPOR encounters two graphs that will yield the same graph immediately after a revisit, then in both cases the revisit happens from the same write w to the same read r, and the graphs only differ in the sets of events that were affected by the revisit (namely, r itself and all the events deleted by the revisit).

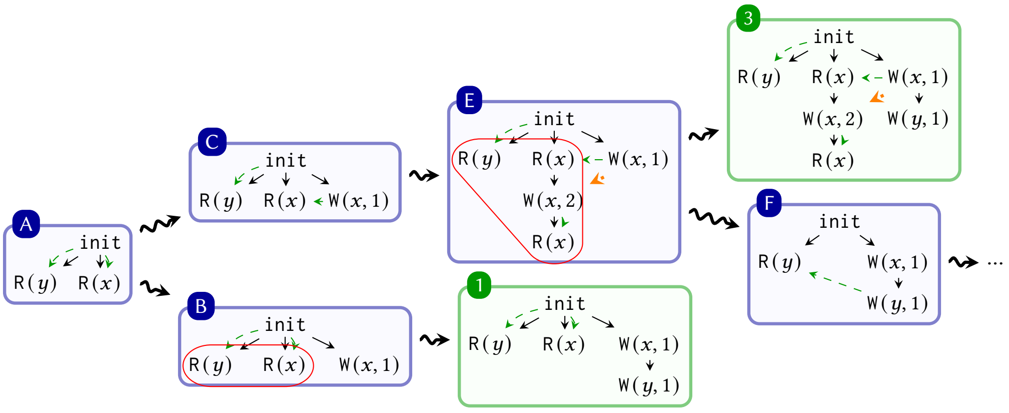
TruSt therefore constrains the events affected by the revisit (i.e., the read being revisited and the deleted events) to form a *maximal extension*: to be added `co`-maximally w.r.t. to the porf-prefix of the revisiting write. Maximal conditions are better understood with an example.

Example 3 Consider the REV-EX below along with its SC-consistent execution graphs.

$$\text{T1: } a := y \;\;\|\;\; \text{T2: } \textbf{if } (x = 1) \; x := 2 \; c := x \;\;\|\;\; \text{T3: } x := 1 \; y := 1 \qquad (\text{REV-EX})$$



A DPOR run producing these execution can be seen below.



Assuming that DPOR adds events in a left-to-right manner, after adding the events of the first two threads, it then adds W$(x, 1)$ which can either revisit R(x) or not (graphs C and B, respectively).

Following the respective subexplorations, W$(y, 1)$ is encountered in both cases: in exploration B immediately, and in exploration C after adding the events under the conditional of T2[1]. Similarly to W$(x, 1)$, in both subexplorations, W$(y, 1)$ has the opportunity to either revisit R(y) or not.

Revisiting R(y) in both cases, however, leads to duplication, as the same graph (graph F) will be obtained twice. Maximal extensions dictate that the revisit only takes place from execution E, as all the affected events are added maximally w.r.t. W$(y, 1)$. To see why, it is helpful to think "backwards": starting from the graph obtained from the revisit without the write and read participating in the revisit (W$(y, 1)$ and R(y)), if all the affected events are added in a co-maximal manner (i.e., reads reading the co-latest write and writes added last in co), we get graph E, which is the graph from where the revisit takes place.

To define maximal extensions, we first introduce an auxiliary definition about execution graphs. A write event w is co-*maximal* in a set of events E if $w \in E$ and it does not have a co-successor in E (i.e., $\nexists w' \in E.\ \langle w, w' \rangle \in \text{co}$).

[1]These events have a unique co and rf option as SC enforces coherence: informally, T2 is already aware of W$(x, 1)$ so W$(x, 2)$ has to be co-after it, and R(x) has to read the latest value T2 is aware of.

Definition 2.1. An event e in a graph G is *added maximally* w.r.t. a write event w in G, if the following conditions hold, where E is the set of all events e' added before e or in w's porf prefix (i.e., $\langle e', w\rangle \in \mathsf{porf}$):

- If $e \in \mathsf{W}$, then e is co-maximal in E.
- If $e \in \mathsf{R}$, then $G.\mathsf{rf}(e)$ is co-maximal in E.

Observe that non-write/read events are always added maximally w.r.t. a revisiting write.

Maximal extensions also have the following useful property, which we will use in some of our examples below.

Proposition 2.2. *If a write w revisits a read r resulting in a graph G, the porf-prefix of w will not be removed in any of the subsequent subexplorations starting from G [Kokologiannakis et al. 2022].*

2.2 Spore: Thread-Level Symmetries

Consider again the W+R+R example where T2 and T3 share their code.

$$\text{T1: } x := 1 \;\Big\|\; \text{T2: } r_2 := x \;\Big\|\; \text{T3: } r_3 := x \qquad (\text{W+R+R})$$

We say that executions $\boxed{2}$ and $\boxed{3}$ from its consistent executions (see Example 1) are symmetric because one can be obtained by permuting the symmetric threads of the other.

2.2.1 Distinguishing Among Symmetric Executions. To avoid exploring both graphs, we pick a *representative* execution among them and instrument DPOR to drop non-representative symmetric executions.

Spore achieves this using thread IDs: we deem as representative the graph where a symmetric thread only reads values that are at least as "recent" (in terms of co) as the ones read by its symmetric predecessor. In the W+R+R example, this means that graph $\boxed{2}$ is the representative one, as in graph $\boxed{3}$ the read of T2 reads a value that is co-after the one read by T3[2].

Let us formalize this intuition. We say that two events e, e' in an execution graph G are *prefix-matching* (and write prefix-matching(e, e')), if they originate from threads with the same code and have matching po-prefixes, i.e., all events po-before them are either not memory accesses or reads that pairwise read from the same write. Note that two writes can be prefix-matching, but any po-later pair of events cannot be: writes break matching prefixes because they are co-ordered.

Spore picks as representative graphs the ones where the thread order of prefix-matching events does not contradict an extension of co called *extended coherence order*: $\mathsf{eco} \triangleq (\mathsf{co} \cup \mathsf{rf} \cup \mathsf{rb})^+$, where $\mathsf{rb} \triangleq \mathsf{rf}^{-1}; \mathsf{co}$ is the *reads-before* order, denoting that a read reads from a write whose value is later overwritten. Observe that, due to the definition of prefix-matching events above, any eco path between two prefix-matching events will involve co.

Given this notion of representative graphs, in the W+R+W example above, graph $\boxed{2}$ in Example 1 is the representative because eco agrees with the thread order (there is an $\mathsf{rb}; \mathsf{rf}$ path from T2 to T3), but graph $\boxed{3}$ is not as eco contradicts the thread order.

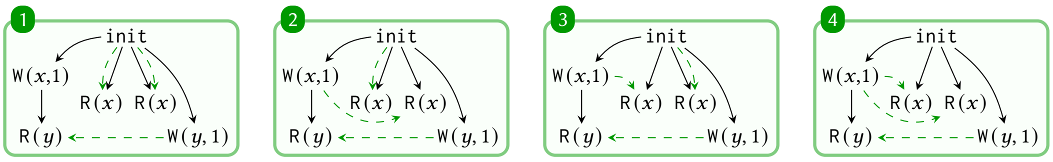
2.2.2 Problem #1: The Interaction Between Representative and Maximal Executions. This solution, however, does not work that easily due to *revisiting* (§2.1). The problem is that SR avoids exploring certain graphs (i.e., the non-representative ones), the exploration of which DPOR might *require* so that a given revisit happens. Put differently, maximal extensions can be non-representative graphs.

[2]Recall that all writes are co-after the initializer event.

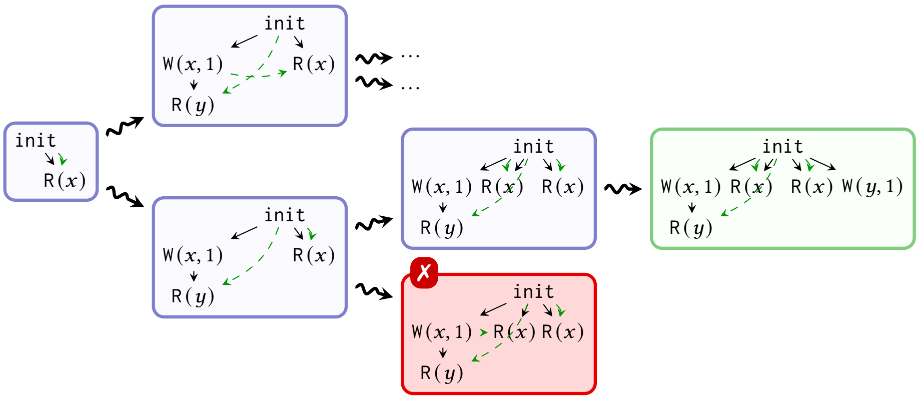
Example 4 To illustrate the problem, consider the following variation of W+R+R (again, T2 and T3 share their code), and suppose we are interested in the executions where $a = 1$.

$$\begin{array}{l||l||l||l}
\text{T1: } x := 1 & \text{T2: } r_2 := x & \text{T3: } r_3 := x & \text{T4: } y := 1 \\
\quad a := y & & &
\end{array} \qquad \text{(W+R+R-REV)}$$

Similarly to W+R+R, graphs 2 and 3 are symmetric, and graph 2 is the representative one.



We now present a (partial) DPOR exploration of this program, with the objective of showing that the combination of DPOR and SR is not guaranteed to be correct. Concretely, we will show that execution 1 will not be generated if DPOR explores the program threads in a peculiar order[3].



Suppose DPOR first adds the read of T3, and then proceeds with the events of T1. When it adds W$(x, 1)$, it can either revisit R(x) (top exploration tree) or not (bottom exploration tree). Since we are interested in generating execution 1, let us disregard the top exploration tree (where T3 reads 1) and focus on the bottom one. (The reason we discard the top one is that DPOR does not "undo" revisits: since W$(x, 1)$ revisits R(x) of T3, in all subsequent subexplorations T3 keep reading 1; see Prop. 2.2.)

At the next step, the algorithm will add the read of T2, which can either read 1 (from T1) or 0 (the initial value). DPOR, however, will only consider the exploration where the read is reading 0, and not the execution where it reads 1, as the latter is not the representative among the symmetric ones. (The one where T2 reads 0 and T3 reads 1 is.)

At the final step, the algorithm will add the W$(y, 1)$ event of T4, and will consider to revisit the R(y). With the maximal extension condition of §2.1, however, this revisit is doomed to fail, since the read of T2 is not added co-maximally w.r.t. W$(y, 1)$. Hence DPOR will not generate execution 1.

[3]DPOR should be able to generate all executions of a program irrespective of the order in which it encounters its threads.

As the W+R+R-REV example demonstrated, the problem when combining DPOR and SR is that resulting algorithm might deem the graphs on which TruSt's maximal extension condition enables a certain revisit as non-representative (and therefore drop them).

There are two potential solutions to this problem.

The first is to modify the maximal extension condition to hold only for representative graphs. Unfortunately, this approach does not work because of the atomicity condition of read-modify-write (RMW) operations. In our technical appendix [Kokologiannakis et al. 2024b], we show that it is impossible to define a maximality condition purely at the level of execution graphs without consulting the program.

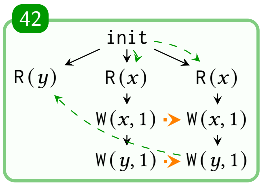
The second solution is to *keep* the maximal extension condition intact, but restrict the *exploration order* so that representative executions always form maximal extensions. To see why restricting the exploration order is a promising solution, let us consider again Example 4. The reason why a maximal extension was created in a non-representative execution was that T3 was added before T2 (i.e., against thread order), and T2 had co-later options available to it (T1 was added after T3 but before T2). By fixing the exploration order, we essentially try to "force" co to agree with the thread order.

2.2.3 Problem #2: Fixing the Exploration Order is Inadequate. Given the above, a natural choice is to maintain a left-to-right scheduling among threads that share their code. Even though this simple modification mitigates the issue in W+R+R-REV, it does not restore correctness in general.

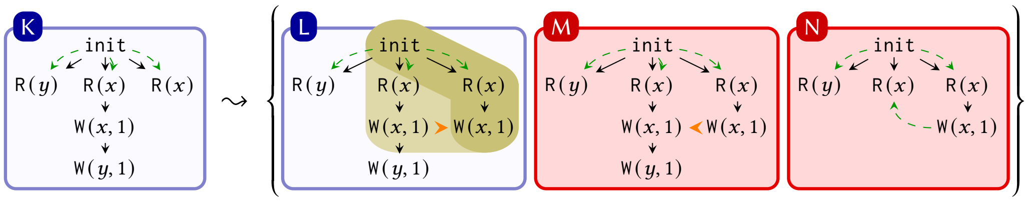
Example 5 To see why, consider the program below where T2 and T3 share their code, along with one of its representative executions.

T1: $a := y$	T2: $r_2 := x$	T3: $r_3 := x$
	$x := 1$	$x := 1$
	$y := 1$	$y := 1$

(R+RWW+RWW)



Assuming that we schedule all threads in a left-to-right manner, execution 42 cannot be generated by the procedure described so far. The first point where the algorithm has more than one choice to consider is the addition of R(x) of T3. The case where R(x) reads from W$(x, 1)$ cannot lead to 42 because the restriction of the graph upon the revisit of R(y) will preserve the rf-edge of the R(x) read. Therefore, we are left with the case where R(x) reads from init (graph K below).



When the W$(x, 1)$ of T3 is added to K, there are three options:

- L: W$(x, 1)$ is added co-after T2's W$(x, 1)$. This execution is explored by DPOR, but cannot lead to the graph 42 because when W$(y, 1)$ is added in T3, it will be unable to revisit R(y) because the W$(x, 1)$ of T2 is not maximally added w.r.t. T3's W$(y, 1)$: it is co-before T3's W$(x, 1)$, which is in T3's porf-prefix.

- M: W$(x, 1)$ is co-before T2's W$(x, 1)$. This execution is dropped because co contradicts thread-order of symmetric events.
- N: W$(x, 1)$ revisits the R(x) of T2. This execution is also dropped because it is not a representative one (T2 is reading a co-earlier value than T3).

As the R+RWW+RWW example above clearly demonstrates, fixing the scheduling policy is insufficient to guarantee completeness. Essentially, the issue described in §2.2.2 still persists: execution 42 could not be produced because a maximal extension was dropped (graph M) in favor of the representative one (graph L). In turn, in the representative execution L, a co-edge from a symmetric thread to the porf-prefix of the revisiting write precluded the revisit.

This last observation is key in marrying DPOR and SR: since a revisit fails due to an event of a symmetric thread being added non-maximally, Spore's solution is to consider symmetric events part of the revisiting write's prefix. In the case of R+RWW+RWW, when Spore considers the revisit between the W$(y, 1)$ of T3 and the R(y) of T1, the prefix of W$(y, 1)$ will include not just the events porf-before it, but also the porf-prefix of symmetric events as well (namely, event W$(x, 1)$ of T2). As such, graph 42 will be generated from L because all the affected events (namely, T1's R(y) and T2's W$(y, 1)$) are added maximally w.r.t. the new prefix of W$(y, 1)$.

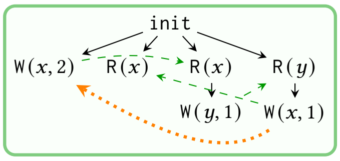
2.2.4 Problem #3: Handling po ∪ rf ∪ co *cycles.* Changing the notion of a prefix is instrumental in restoring completeness, but comes with a caveat. In DPOR, a write can never revisit events in its own prefix. So, by introducing a new notion of a prefix (henceforth sprefix) in Spore, do we lose any executions? Is it possible that this novel notion of a prefix precludes some revisit that does not create a causal cycle, thereby rendering Spore incomplete?

The answer depends on the underlying memory model. First, we can show that sprefix cycles boil down to po ∪ rf ∪ co cycles. (Our full argument is presented in §3.) Strong models, such as SC, TSO [SPARC International Inc. 1994], and SRA [Lahav et al. 2016], forbid $(\text{po} \cup \text{rf} \cup \text{co})^+$ cycles, and so it is never possible for a read to read from a write in its sprefix.

In weaker models, such as RC11 [Lahav et al. 2017], however, the answer is yes: it *can* be the case that an event is in its own sprefix but not in its own porf-prefix. Such a scenario is shown below.

Example 6 Consider the SP-CYC program, where T2 and T3 share their code.

```
T1: x := 2  ||  T2: r2 := x        ||  T3: r3 := x        ||  T4: a := y
            ||      if (r2 = 2)    ||      if (r3 = 2)    ||      x := 1
            ||        y := 1       ||        y := 1       ||
                                                               (SP-CYC)
```



In the execution of Example 6, W$(x, 1)$ is in its own sprefix (W$(x, 1)$ is read from the R(x) of T2, which is symmetric to the R(x) of T3, which is in turn in the prefix of W$(x, 1)$), but not in its own porf-prefix (there is no porf cycle).

To restore completeness, Spore therefore checks that no consistent execution graph has a po ∪ rf ∪ co cycle. This condition typically holds: a po ∪ rf ∪ co cycle implies that there exist two writes that are not porf-ordered, and such unordered concurrent writes are rare in realistic implementations [Abdulla et al. 2019; Kokologiannakis et al. 2019b]. As we show in §4, Spore is directly applicable to realistic libraries of concurrent data structures.

```
enqueue($v$) ≜
  $node$ := malloc(...)
  $node.value$ := $v$
  $node.next$ := NULL
  do
    $t$ := $\underline{tail}$
    $next$ := $t.next$
    if ($t \neq \underline{tail}$) continue
    if ($next \neq$ NULL)
      $\boxed{\text{CAS}(\underline{tail}, t, next)}$
      continue
  while ($\neg$CAS($t.next$, $next$, $node$))
  $\boxed{\text{CAS}(\underline{tail}, t, node)}$
```

```
dequeue() ≜
  do
    $h$ := $\underline{head}$
    $n$ := $h.next$
    if ($h \neq \underline{head}$) continue
    if ($n$ = NULL) return None
  while ($\neg$CAS($\underline{head}$, $h$, $n$))
  $t$ := $\underline{tail}$
  if ($h = t$)
    $\boxed{\text{CAS}(\underline{tail}, t, n)}$
  $v$ := $n.value$
  reclaim($h$)
  return $v$
```

```
rdcss_read($a_2$) ≜
  $r$ := $a_2$
  while (is_desc($r$))
    $\boxed{\text{complete}(r)}$
    $r$ := $a_2$
  return $r$
```

```
complete($d$) ≜
  $r$ := $d.a_2$
  $n$ := ($r = d.o_1$) ?
          $d.n_2$ : $d.o_2$
  CAS($d.a_2$, $d$, $n$)
```

```
rdcss($d$) ≜
  $r$ := CAS($d.a_2$, $d.o_2$, $d$)
  while (is_desc($r$))
    $\boxed{\text{complete}(r)}$
    $r$ := CAS($d.a_2$, $d.o_2$, $d$)
  if ($r = d.o_2$) $\boxed{\text{complete}(d)}$
  return $r$
```

Fig. 2. DGLM queue (left) and RDCSS (right). Global variables are underlined; function arguments are passed by reference; CAS returns whether it succeeded.

2.3 Spore: Internal Symmetries

We now switch gears and present how Spore exploits internal symmetries. We first present some examples of such symmetries (§ 2.3.1), and then discuss Spore's treatment (§ 2.3.2). We end this section by discussing how internal and thread-level symmetries interact (§ 2.3.3).

2.3.1 Internal Symmetry Examples. Fig. 2 shows two examples of internal symmetries: the Doherty-Groves-Luchangco-Moir (DGLM) queue [Doherty et al. 2004] and Restricted Double-Compare Single Swap (RDCSS) [Harris et al. 2002].

DGLM queue is a lock-free queue comprising two pointers *head* and *tail*. At the end of each enqueue operation, each enqueuer advances the *tail* pointer to point to the last element of the queue. If, however, a concurrent enqueuer or dequeuer detects that the *tail* pointer is lagging behind (i.e., $tail.next \neq$ NULL), it tries to advance *tail* on behalf of an incomplete enqueue.

RDCSS is a double CAS operation that takes as an argument a descriptor d containing two addresses a_1, a_2 with their expected values o_1, o_2 and a new value n_2. If both addresses contain their expected values, then the new value n_2 is stored at the second address a_2. To perform the double comparison atomically, RDCSS first tries to place its descriptor in the a_2 address, and then reads a_1 to determine whether to replace it with the new value n_2 or restore the old value o_2. In case another thread encounters the descriptor, it tries to complete the ongoing RDCSS call.

Both algorithms employ the textbook *helping pattern* [Herlihy 1991; Herlihy and Shavit 2008], where some operation A observes an ongoing, incomplete operation B and tries to complete B before performing its own. This helping pattern appears ins widely used concurrent libraries, including `libcds` [Khizhinsky n.d.], `folly` [Facebook n.d.] and `ckit` [Bahra n.d.], as well as in most algorithms described by Herlihy and Shavit [2008];

Observe that in both cases, the highlighted $\boxed{\textit{main}}$ and $\boxed{\textit{helping}}$ operations are *idempotent*: one of the CASes succeeds and all the others fail without changing the state. Moreover, their result is the same irrespective of which operation succeeds, and that the program cannot distinguish *which* operation succeeded. Indeed: (i) both operations execute exactly the same code, (ii) their returned value is not checked by the program, and (iii) swapping which of the operations succeeded preserves consistency and does not mask any error. As we will shortly see, these three conditions enable Spore to exploit internal symmetries and drastically reduce the state space. (In contrast, thread-level symmetries are inapplicable because the main and the helping operations have different execution prefixes.)

2.3.2 Exploiting Idempotent Operations. Spore exploits idempotent operations by only exploring executions where the $\boxed{\textit{main}}$ operation succeeds. To this end, Spore changes the underlying memory model and treats $\boxed{\textit{helping}}$ operations as no-ops, which have no incoming/outgoing `rf` or `co` edges. To do that, Spore requires assistance from the user: the user annotates helping operations in the program (as in Fig. 2), and then Spore automatically treats them as no-ops and reduces the state space to be searched.

Annotations bring us to a major challenge that needs to be resolved: ensuring annotation correctness. If users incorrectly annotate a function as helping, it might mask an existing error in the user program. As such, Spore uses a *dummy event* in the place of the function to check whether certain (sufficient) conditions hold. If they do not, Spore reports an annotation error to the user.

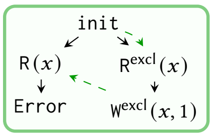
Some minimal preconditions that need to hold for a function f_h to be considered as helping w.r.t. a function f_m have already been stated in §2.3.1: (i) f_h and f_m execute the same code, (ii) the returned value of f_h and f_m is not checked by the program, and (iii) replacing an execution where f_m fails and f_h succeeds with one where f_m succeeds and f_h is treated as no-op preserves consistency and the presence of an error.

Let us now go over these conditions in more detail. The first two conditions lie at the heart of idempotency, and are what allow Spore to treat f_h as a no-op: no code uses the result of f_h and is thus safe to disregard it. Had f_h and f_m been different (or had their results been used), then annotating one of them as helping would mask errors in programs, like in the example below.

Example 7 Consider the helper-cf program, along with one of its execution graphs.

T1: $\boxed{a := \mathtt{CAS}(x, 0, 1)}$
    $\mathtt{assert}(a = 0)$
|| T2: $\boxed{\mathtt{CAS}(x, 0, 1)}$

(helper-cf)



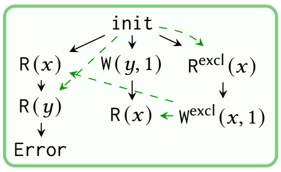
f_m and f_h are functions comprising a single CAS operation, but the result of f_m is used (i.e., f_h is incorrectly annotated as helping). If we treat f_h as a dummy event, the execution above (where the failed CAS generates a single read event and the successful one two events annotated with an excl flag) will not be explored and the error will be missed.

Condition (iii) is a bit more intricate. To ensure it, we need to guarantee that in any execution where f_h succeeds, f_m has already observed (in a synchronizing manner) the operations of f_h. If reading from writes in f_m can imply less synchronization with the rest of the program, then it is possible that reading from f_h results in an error, but reading from f_m does not (and thus, treating f_h as dummy can mask errors). We demonstrate this point with the following example.

Example 8 Consider the helper-sync program under SC.

T1: $a := x$
    if $(a = 1)$
        $\mathtt{assert}(y = 1)$
|| T2: $y := 1$
    $\boxed{\mathtt{CAS}(x, 0, 1)}$
|| T3: $\boxed{\mathtt{CAS}(x, 0, 1)}$

(helper-sync)



If the CAS in T2 succeeds and T1's read of x reads from it, then T1 will necessarily read $y = 1$. If, however, the CAS in T3 succeeds and T1 reads from it (as shown in the graph above), T1 can subsequently read $y = 0$ and violate its assertion (as shown in the graph above).

To fix this last issue, Spore imposes four more conditions on the user annotations:

(1) f_m and f_h have no other writes apart from a final CAS
(2) f_m has a preceding *source event* whose value it uses as the compare operand
(3) f_m is immediately preceded by a write, which is observed in a synchronizing manner before f_h
(4) all writes to the location of f_m's CAS are part of *read-modify-write* (RMW) operations

These conditions are formalized in §3. As we prove in §3, these conditions are sufficient to detect erroneously annotated helping patterns.

2.3.3 The Interaction Between Internal and Thread-Level Symmetries. Before moving on to our formal discussion of Spore, it is worth noting that idempotent operations *facilitate* SR. Consider an example with two symmetric threads performing a helping CAS. Assuming that the threads are symmetric up until the CASes, treating the CASes as an RMW operations breaks the symmetry, while treating them as dummy events preserves the symmetry.

3 Spore: Formal Description

In this section, we describe the theoretical basis of Spore. In particular, we explain: (§ 3.1) the representation of executions as execution graphs; (§3.2) how Spore can be represented as a memory model; (§3.3) Spore's exploration algorithm; (§3.4) why Spore is correct, i.e., why it explores exactly one graph per the combined equivalence classes of DPOR and SR, and does not mask any errors.

3.1 Execution Graphs

An execution graph comprises a set of events (nodes), and a few relations on these events (edges).

Definition 3.1. An *event*, $e \in \mathsf{Event}$, is either the initialization event `init`, or a thread event $\langle t, i, l\rangle$ where $t \in \mathsf{Tid}$ is a thread identifier, $i \in \mathsf{Idx}$ is a serial number (denoting the index of an event within a thread), and $l \in \mathsf{Lab}$ is a label that takes (at least) one of the following forms:

- Write label: $\mathtt{W}^k(l, v) \in \mathtt{W}$, where k records the write attributes, $l \in \mathsf{Loc}$ the location accessed, and $v \in \mathsf{Val}$ the value written.
- Read label: $\mathsf{R}^k(l, v) \in \mathsf{R}$, where k records the read attributes, $l \in \mathsf{Loc}$ the location accessed, and $v \in \mathsf{Val}$ the value read.
- Annotated function label: $\mathtt{M}^m(f, as) \in \mathtt{M}$, where $m \in \{\mathsf{main}, \mathsf{help}\}$ is the function attribute, $f \in \mathsf{Fname}$ is the name of the function been called, and $as \in \mathsf{Val}^*$ is a sequence representing the function arguments.

Read and write attributes include the exclusivity flag excl for RMWs, and the access mode for RC11-style models. (Additional kinds of events exist for memory allocations, deallocations, assertion violations, *etc.*, but these do not affect the model checking algorithm in any meaningful way.)

Having defined events, we define execution graphs as follows.

Definition 3.2. An *execution graph* $G \in \mathsf{Exec}$ comprises the following components:

(1) a set of events E that includes `init` and does not contain multiple events with the same thread identifier and serial number;
(2) $\mathtt{rf} : E \cap \mathsf{R} \to E \cap \mathtt{W}$, called the *reads-from* function, mapping each read event to a same-location write from where it gets its value;

(3) $\mathsf{co} \subseteq \bigcup_{l \in \mathrm{Loc}} \mathtt{W}_l \times \mathtt{W}_l$ (where $\mathtt{W}_l \triangleq \{\mathtt{init}\} \cup \{\langle t, i, l\rangle \in E \mid l = \mathtt{W}\text{-}(l, \_)\}$) called the *coherence order*, a strict partial order that is total on $\mathtt{W}_l$ for every location $l \in \mathrm{Loc}$; and
(4) $\leq$, a total order on E that represents the order in which events were incrementally added to the graph.

Conventions

We write $G.\mathsf{E}$, $G.\mathtt{rf}$, $G.\mathsf{co}$ and $\leq_G$ to project the various components of an execution graph. Given two events $e_1, e_2 \in G.\mathsf{E}$, we write $e_1 <_G e_2$ if $e_1 \leq_G e_2$ and $e_1 \neq e_2$. In relational algebra expressions, we abuse notation and write $G.\mathtt{rf}$ for the relation $\{\langle G.\mathtt{rf}(r), r\rangle \mid r \in G.\mathsf{R}\}$.
We assume that $\mathtt{init} \in \mathtt{W}$, and omit the $\emptyset$ for read/write labels with no attributes.
The functions $\mathtt{tid}$, $\mathtt{idx}$, $\mathtt{loc}$, $\mathtt{mod}$ and $\mathtt{arg}$ respectively return the thread identifier, serial number, location, access mode and function arguments of an event, when applicable.
We write $G.\mathtt{W}$ for $G.\mathsf{E} \cap \mathtt{W}$ (and similarly for other sets), and use superscript and subscripts to restrict label sets (e.g., $\mathtt{W}_l \triangleq \{\mathtt{init}\} \cup \{w \in \mathtt{W} \mid \mathtt{loc}(w) = l\}$).

Observe that G does not have an explicit *program order* (po) component. We induce po based on our representation of events as follows:

$$\mathsf{po} \triangleq \left\{\langle \mathtt{init}, e\rangle \;\middle|\; e \in \mathsf{Event} \setminus \{\mathtt{init}\}\right\} \cup \left\{\langle\langle t_1, i_1, l_1\rangle, \langle t_2, i_2, l_2\rangle\rangle \;\middle|\; t_1 = t_2 \wedge i_1 < i_2\right\}$$

In our technical appendix [Kokologiannakis et al. 2024b], we define two mappings from programs to sets of execution graphs: (1) $[\![.]\!]$, which ignores function annotation labels, and simply generates an event with a $\mathtt{M}^m$ label before the events corresponding to the function body; and (2) $[\![.]\!]^{\mathsf{Annot}}$, which in the case of functions annotated with help, generates only the $\mathtt{M}^{\mathsf{help}}$ event and does not generate any events for the body of the function call. Both mappings keep the $\mathtt{rf}$ and co components of graphs completely unconstrained. These components will be constrained by the memory model.

3.2 Consistency and Error Detection

A memory model, M, comprises three components: (a) a *causal prefix* relation, cb_M, (b) a *consistency predicate* $\mathsf{consistent}_\mathsf{M}(G)$ that determines whether an execution graph G is consistent, and (c) an $\textsc{IsErroneous}_\mathsf{M}(G)$ predicate, prescribing whether G contains an error (e.g., an invalid memory access) according to M.

The consistency predicate is used to constrain the semantics of a program. The annotation-ignoring (resp. annotation-aware) semantics of a program P under a memory model M, denoted $[\![P]\!]_\mathsf{M}$ (resp. $[\![P]\!]^{\mathsf{Annot}}_\mathsf{M}$), is given by the set of execution graphs in $[\![P]\!]$ (resp. $[\![P]\!]^{\mathsf{Annot}}$) that are M-consistent.

In Spore, we assume an underlying memory model M with $\mathsf{cb}_\mathsf{M} = (\mathsf{po} \cup \mathtt{rf})^+$, $\mathsf{consistent}_\mathsf{M}(\cdot)$ being *extensible*, *prefix-closed*, and implying RMW atomicity and cb_M-acyclicity [Kokologiannakis et al. 2022], and $\textsc{IsErroneous}_\mathsf{M}(\cdot)$ being *prefix-monotone*. Models satisfying these requirements include SC [Lamport 1979], TSO [SPARC International Inc. 1994], Release-Acquire (RA) [Lahav et al. 2016], and RC11 [Lahav et al. 2017]. We then define a new memory model, SYM, with

$$\begin{aligned}
\mathsf{cb}_{\mathsf{SYM}} &\triangleq (\mathsf{po} \cup \mathtt{rf} \cup \mathsf{symb})^+ \\
\mathsf{consistent}_{\mathsf{SYM}}(G) &\triangleq \mathsf{consistent}_\mathsf{M}(G) \wedge \textsc{irreflexive}(\mathsf{symb}; \mathsf{eco}) \\
\textsc{IsErroneous}_{\mathsf{SYM}}(G) &\triangleq \textsc{IsErroneous}_\mathsf{M}(G) \vee \neg\textsc{irreflexive}((\mathsf{po} \cup \mathtt{rf} \cup \mathsf{co})^+) \\
&\quad \vee G \text{ is incorrectly annotated (see Def. 3.3 below)}
\end{aligned}$$

where $G.\mathsf{symb}$ is the *symmetry-before* order that orders prefix-matching events according to their thread order. Concretely, $\langle e_1, e_2\rangle \in G.\mathsf{symb}$ if the following hold:

(i) $\texttt{idx}(e_1) = \texttt{idx}(e_2)$ and $\texttt{tid}(e_1) < \texttt{tid}(e_2)$
(ii) e_1 and e_2 originate from threads running the same code (and spawned consecutively),
(iii) have no preceding same-thread writes, and
(iv) for every preceding same-thread read r_1 of e_1, the corresponding (i.e., having the same index) read r_2 in $\texttt{tid}(e_2)$ has the same $\texttt{rf}$ (i.e., $G.\texttt{rf}(r_1) = G.\texttt{rf}(r_2)$).

Annotation Correctness. To ensure annotation correctness, Spore first checks that for each $f_h \in G.\mathsf{M}^{\text{HELP}}$, there exists a (unique) $f_m \in G.\mathsf{M}^{\mathsf{main}}$ with the same arguments, and that these functions do not return any results (cf. conditions (i) and (ii) of §2.3.2), and are well-formed:.they comprise a (possibly empty) sequence of reads followed by a CAS operation, with a possible data dependency from the reads to the CAS (no other dependencies are allowed so that the locations accessed can be deduced by the arguments of f_m/f_h).

Assuming both functions has the proper form, Spore has to now ensure that (iii) holds, i.e., that their synchronization is the same. Since the definition of synchronization differs among memory models, for simplicity, we here provide a definition that works for SC and RA[4]. In what follows, we lift $\texttt{loc}/\texttt{exp}$ to return the location/expected-value of the CAS read following an $f_m \in G.\mathsf{M}^{\mathsf{main}}$.

Our definition uses the notion of a *source write* s at location $\texttt{loc}(f_m)$, which is observed before f_m (i.e., either it is po-before f_m or it is read po-before f_m), and writes the value $\texttt{exp}(f_m)$. We also require that the immediate po-predecessor of f_m is observed before f_h, which ensures that the f_h has synchronized with everything in f_m's prefix, and that all writes to $\texttt{loc}(f_m)$ after s are RMWs and do not write the same value as s. The latter condition ensures that f_m and f_h cannot both succeed, and that if f_h succeeds, then f_m observes its update.

Definition 3.3 (Annotation correctness). An execution G is *correctly annotated* if for all $f_h \in G.\mathsf{M}^{\mathsf{help}}$, there exist (a) a corresponding $f_m \in G.\mathsf{M}^{\mathsf{main}}$ with $\texttt{arg}(f_m) = \texttt{arg}(f_h)$ and (b) a source write $s \in G.\mathsf{W}$ with $\texttt{loc}(s) = \texttt{loc}(f_m)$ and $\texttt{val}(s) = \texttt{exp}(f_m)$ such that:

- $\langle s, f_m \rangle \in G.\texttt{rf}^?;\mathsf{po}$ (s is observed before f_m),
- $\langle f_m, f_h \rangle \in \mathsf{po}^{-1}|_{\text{imm}}; G.\texttt{rf}; \mathsf{po}$ (the immediate predecessor of f_m is observed before f_h),
- for all $w \in rng([s]; \texttt{co})$, $w \in \mathsf{W}^{\mathsf{excl}}$ and $\texttt{val}(w) \neq \texttt{val}(s)$ (all subsequent writes to $\texttt{loc}(f_m)$ are RMWs and write different values).

3.3 Exploration Algorithm

Let us now proceed by showing how Spore enumerates all SYM-consistent execution graphs of a program P. The algorithm is shown in Algorithm 1, which constructs the consistent graphs incrementally by recording the event addition order in the graphs' $\leq_G$ component. Spore is optimal in the sense that it only explores consistent execution graphs and it never explores two execution graphs that differ only in their $\leq_G$ components.

Spore verifies the input program P under a memory model M by calling Explore with the initial graph $G_\emptyset$ containing only the initialization event $\texttt{init}$.

First, Explore(P, G) checks whether the current graph contains an error (Line 2). Note that errors are checked against Spore's memory model: they include not only errors under the underlying memory model M, but also *user annotation errors*.

In addition, recall that Spore's errors include the existence of $\mathsf{po} \cup \texttt{rf} \cup \texttt{co}$ cycles. Such a check is necessary to justify why exploring $\texttt{cb}_{\mathsf{SYM}}$-acyclic execution graphs suffices: any $(\mathsf{po}\cup\texttt{rf}\cup\texttt{co})$-acyclic graph where the symmetry-before order does not contradict the $\texttt{eco}$ order is also $\texttt{cb}_{\mathsf{SYM}}$-acyclic.

[4]In our technical appendix [Kokologiannakis et al. 2024b], we provide the definition for the RC11 memory model. The definition for SC/RA is a special case of the RC11 definition.

Algorithm 1 SPORE: An optimal combination of DPOR and SR

```
1:  procedure EXPLORE_P(G)
2:      if ISERRONEOUS_SYM(G) then exit("Error")
3:      $a \leftarrow$ ADDNEXTEVENT_P(G)
4:      if $a \in \mathsf{R}$ then
5:          for $w \in G.\mathsf{W}_{\mathtt{loc}(a)}$ do EXPLOREIFCONSISTENT_P(SetRF(G, a, w))
6:      else if $a \in \mathsf{W}$ then
7:          EXPLORECOs_P(G, a)
8:          for $r \in G.\mathsf{R}_{\mathtt{loc}(a)}$ such that $\langle r, a\rangle \notin G.\mathsf{cb}_{\mathsf{SYM}}$ do
9:              $Deleted \leftarrow \{e \in G.\mathsf{E} \mid r <_G e <_G a \land \langle e, a\rangle \notin G.\mathsf{cb}_{\mathsf{SYM}}\}$
10:             if SHOULDREVISIT(G, ⟨r, a, Deleted⟩) then
11:                 EXPLORECOs_P(SetRF($G \setminus Deleted$, r, a), a)
12:     else if $a \neq \bot$ then
13:         EXPLORE_P(G)
14: procedure EXPLOREIFCONSISTENT_P(G)
15:     if consistent_SYM(G) then EXPLORE_P(G)
16: procedure EXPLORECOs_P(G, a)
17:     for $w_p \in G.\mathsf{W}_{\mathtt{loc}(a)}$ do EXPLOREIFCONSISTENT_P(SetCO(G, $w_p$, a))
```

If the graph is error-free, EXPLORE extends it by one event a from the program by calling ADDNEXTEVENT (Line 3). If there are no events to add, then a full execution of P has been explored, and EXPLORE returns.

If a is a read, then EXPLORE recursively explores all consistent $\mathtt{rf}$ options for that read. As such, for each same-location write w, EXPLORE recursively calls itself (via the helper function EXPLOREIFCONSISTENT) on the graph that results if a reads from w (Line 5). EXPLOREIFCONSISTENT checks whether G is consistent (Line 15), and if so calls EXPLORE recursively. (Recall that consistency also requires that the graph does not violate our SR principle.)

If a is a write, SPORE proceeds with the non-revisit case and the revisit case, respectively. For the non-revisit case, EXPLORE checks for all possible placements of the newly added write in $\mathtt{co}$ by means of EXPLORECOs (Line 7).

For the revisit case, SPORE also checks whether any of the existing reads of G can be *revisited* to read from a: since a was not present when their possible reads-from options were examined, EXPLORE explores these additional $\mathtt{rf}$ options now. Thus, for each same-location read r that does not precede a, if revisiting r will not lead to a duplicate exploration (checked by SHOULDREVISIT[5]), EXPLORE calls EXPLORECOs on the graph that occurs if all the events that were added after r are deleted, excluding a and its predecessors (Line 11).

Observe, however, that as we motivated earlier in §2.2.4, SPORE only explores $\mathsf{cb}_{\mathsf{SYM}}$-acyclic execution graphs. As such, SPORE never revisits reads that are in $\mathsf{cb}_{\mathsf{SYM}}$-before a (as opposed to cb_{M}-before a), as revisiting such reads would create $\mathsf{cb}_{\mathsf{SYM}}$ cycles (the $\mathsf{cb}_{\mathsf{SYM}}$-prefix of a revisiting write is always preserved).

If a has any other type (Line 13), EXPLORE recursively calls itself.

[5]As the definition of SHOULDREVISIT is unnecessary for this discussion, we omit it; we refer interested readers to our technical appendix [Kokologiannakis et al. 2024b].

Remark 1. Observe that, with the exception of annotation errors, Spore does not take any special care for method annotation labels M. Indeed, this is because these are handled implicitly by the interpreter: Line 3 adds events according to our annotated semantics $[\![P]\!]^{\mathsf{Annot}}$. When the interpreter encounters a function annotated with main, it will yield an $\mathsf{M}^{\mathsf{main}}(as)$ (which is not treated specially) as well as the events of the function, while for a function annotated with help it will only yield an $\mathsf{M}^{\mathsf{help}}(as)$ event.

Remark 2. We assume that the AddNextEvent procedure (Line 3), always picks the leftmost thread among the ones that are symmetric, i.e., their next events are prefix-matching. This is necessary for the algorithm's correctness, which demands that when an event e is added, its $\mathsf{cb}_{\mathsf{SYM}}$-prefix already be present in the graph.

3.4 Soundness, Completeness and Optimality

3.4.1 Soundness of Internal Symmetries. We show that if a program P is erroneous under its standard interpretation $[\![P]\!]$ (which ignores annotations), then it is also erroneous under the annotated interpretation $[\![P]\!]^{\mathsf{Annot}}$ (which encodes annotated functions with dummy events). See [Kokologiannakis et al. 2024b] for how programs are mapped to execution graph sets.

Theorem 3.4. *Let P be an annotated program and $G \in [\![P]\!]_M$ such that IsErroneous$_M(G)$. Then, there exists $G' \in [\![P]\!]_M^{\mathsf{Annot}}$ such that IsErroneous$_{SYM}(G)$.*

Proof sketch. It suffices to show that there exists a corresponding execution G' (where every f_h being treated as a (single) dummy event $\mathsf{M}^{\mathsf{help}}(...)$) such that (1) IsErroneous$_{\mathsf{M}}(G')$ holds, or (2) G' is incorrectly annotated (see Def. 3.3). The lack of an annotation error is essential in showing that changing G' so that f_m succeeds instead of f_h does not affect G's consistency.

The conditions of Def. 3.3 essentially enforce that in any execution where f_h would succeed, (a) there is an f_m, running the same code, (b) f_m fails (there can only be one write that writes the expected value), (c) f_m reads from the CAS of f_h, or from a co-later (due to coherence and the presence of the source event), and therefore there is a porf-path from the CAS of f_h to the CAS of f_m (all writes to the CAS location are part of an RMW, and thus such a co path is also a porf path), (d) f_m is preceded by a write that was observed by the thread of f_h. This guarantees that swapping the events of f_m with those of f_h, and replacing the events of f_h with a no-op, adds no synchronization in the execution, and therefore preserves both consistency and the presence of an error.

If any of the previous conditions fails, we show that there exists an execution with f_h being treated as a no-op that is not correctly annotated. □

3.4.2 Correctness of Spore. To state our desired result, we first need to formally define which are the execution graphs that are considered equivalent up to symmetry. Given a program P with N threads, a *valid thread permutation* π is a bijection $\{1, \ldots, N\} \mapsto \{1, \ldots, N\}$ such that threads $\pi(i)$ and i share the same code for all $1 \leq i \leq N$. We say that two executions G_1 and G_2 are *symmetric*, denoted $G_1 \approx G_2$, if there exists a valid thread permutation π such that $\pi(G_1) = G_2$, where $\pi(G_1)$ applies the permutation to all the thread IDs in the events of G_1.

The following proposition demonstrates that the class of M-consistent execution graphs up to symmetry corresponds (one-to-one) to the class of SYM-consistent execution graphs.

Proposition 3.5. *Given a program P and an execution graph $G \in [\![P]\!]_M^{\mathsf{Annot}}$, there is a unique execution graph $G' \in [\![P]\!]_{SYM}^{\mathsf{Annot}}$ such that $G \approx G'$.*

Proof. To obtain G' from G, sort the threads running the same function by the eco of the respective events (lexicographically, in po order). It is easy to see that this ordering is well-defined (there are no cycles), and unique: any possibly eco-unordered threads are in fact equal, and that the constructed graph G' satisfies irreflexive(symb; eco). □

Correctness of the exploration algorithm follows by adapting the proof of Awamoche [Kokologiannakis et al. 2023] and is captured by the following proposition.

Proposition 3.6 (Algorithmic Correctness and Optimality).

(1) $\textsc{Explore}_P(G_\emptyset)$ *terminates.*
(2) $\textsc{Explore}_P(G_\emptyset)$ *only explores* cb_{SYM}*-prefixes of executions in* $[\![P]\!]^{\mathsf{Annot}}_{SYM}$*.*
(3) $\textsc{Explore}_P(G_\emptyset)$ *explores every execution* $G \in [\![P]\!]^{\mathsf{Annot}}_{SYM}$ *such that* $\textsc{irreflexive}(G.\mathsf{cb}_{SYM})$*.*
(4) $\textsc{Explore}_P(G_\emptyset)$ *never explores the same* G *twice.*

Termination holds because either a revisit step is performed and the part of the graph that cannot be changed grows or a non-revisit step is performed and the execution graph grows. Soundness holds by construction because consistency is checked before every recursive call. Completeness is more elaborate: it holds because all possible rf/co options are considered for each newly added event, and moreover previous reads can be revisited in their maximal extension (which always exists and is consistent). Optimality holds because there cannot be two steps leading to the same graph; in case of revisits, that is precluded by the uniqueness of maximal extensions.

We next show that if $[\![P]\!]^{\mathsf{Annot}}_{\mathsf{SYM}}$ includes a $\mathsf{cb}_{\mathsf{SYM}}$-cyclic execution, which the algorithm would not explore, then it also includes a $\mathsf{cb}_{\mathsf{SYM}}$-acyclic execution with a $\mathsf{po} \cup \mathsf{rf} \cup \mathsf{co}$ cycle, which the algorithm would explore and report.

Proposition 3.7 ($\mathsf{cb}_{\mathsf{SYM}}$ cycle). *If there is an execution* $G \in [\![P]\!]^{\mathsf{Annot}}_{SYM}$ *with a* $G.\mathsf{cb}_{SYM}$ *cycle, then there is an execution* $G' \in [\![P]\!]^{\mathsf{Annot}}_{SYM}$ *such that* $\textsc{irreflexive}(G'.\mathsf{cb}_{SYM})$ *and* G' *has a* $\mathsf{po} \cup \mathsf{rf} \cup \mathsf{co}$ *cycle.*

Combining Prop. 3.5, Prop. 3.6(3), and Prop. 3.7, we obtain our completeness result.

Theorem 3.8 (Completeness). *If there exists* $G \in [\![P]\!]^{\mathsf{Annot}}_{SYM}$ *such that* $\textsc{IsErroneous}_{SYM}(G)$*, then* $\textsc{Explore}_P(G_\emptyset)$ *will report an error. Otherwise, for each* $G \in [\![P]\!]^{\mathsf{Annot}}_{M}$*,* $\textsc{Explore}_P(G_\emptyset)$ *will explore an execution* $G' \in [\![P]\!]^{\mathsf{Annot}}_{SYM}$ *such that* $G \approx G'$*.*

Combining Prop. 3.5 and Prop. 3.6(4), we obtain our optimality result.

Theorem 3.9 (Optimality). *For any two executions* G *and* G' *explored by* $\textsc{Explore}_P(G_\emptyset)$*,* $G \not\approx G'$*.*

4 Evaluation

We implemented Spore as a tool for C/C++ programs on top of the open-source GenMC stateless model checker, which implements the TruSt algorithm for DPOR. We reused GenMC's infrastructure for interpreting programs and constructing and maintaining execution graphs, but replaced GenMC's consistency checking and error detection mechanism with the ones described in §3.1. We also modified the notion of a prefix used in graph construction to use $\mathsf{cb}_{\mathsf{SYM}}$, and made GenMC's scheduler respect $\mathsf{cb}_{\mathsf{SYM}}$ when encountering symmetric threads.

4.1 Goals

We evaluate Spore on a set of real-world implementations with two goals: (1) show that Spore scales well enough to verify useful implementations (and determine its scalability limit), and (2) determine to what extent its scalability should be attributed to internal vs thread-level symmetries.

To attain these goals, we run Spore on a set of representative real-world clients and benchmarks. The clients evaluate the effectiveness of the SR algorithm, while the benchmarks evaluate the effectiveness of Spore's modeling of internal symmetries. To further study how internal and thread-level symmetries contribute to Spore's performance, we compares Spore against (a) plain SMC enhanced with SR (SR), (b) a baseline TruSt implementation (TruSt), (c) Spore without thread-level symmetries (DPOR+IS), and (d) Spore without internal symmetries (DPOR+SR). Our evaluation is performed under RC11.

As we show, Spore yields a huge improvement over the state-of-the-art as it can gracefully scale to up to 6 threads (often to many more), and both internal and thread-level symmetries are crucial for its scalability to more threads.

Experimental Setup. We conducted all experiments on a Dell PowerEdge R6525 system running a custom Debian-based distribution with 2 AMD EPYC 7702 CPUs (256 cores @ 2.80 GHz) and 2TB of RAM. We set the timeout limit to 30 minutes (denoted by ⏲). All times are in seconds.

We also ran some of our benchmarks against the DPOR implementation of Nidhugg [Abdulla et al. 2014], which obtained similar and/or worse results than TruSt (see [Kokologiannakis et al. 2024b]).

4.2 Benchmarks

To evaluate the effectiveness of thread-level symmetries, we used three different clients:

- Multiset(N): $\lceil \frac{N}{2} \rceil$ (resp. $\lfloor \frac{N}{2} \rfloor$) threads insert (resp. remove) elements at a data structure; the client checks whether each removed element was previously inserted.
- LIFO/FIFO(N): two threads check for the LIFO/FIFO property, while $\lceil \frac{N}{2} \rceil$ (resp. $\lfloor \frac{N}{2} \rfloor$) threads create "noise" in the queue to increase traffic, by inserting (resp. removing) elements.
- Empty(N): N threads insert an element and subsequently remove an element; the client ensures each removal succeeds.

As it can be seen, the clients become progressively more challenging in the sense that the number of multiple operations per thread increases, which hinders symmetry reduction.

To demonstrate that Spore is applicable to non-data-structure benchmarks as well, we used two other clients (Fig. 4):

- Mutex(N): N threads perform a lock followed by an unlock operation.
- RDCSS(N): N threads perform an RDCSS call followed by an RDCS/read call, and 2 threads perform a single RDCSS call.

To evaluate the effectiveness of internal symmetries, we used some representative benchmarks both with and without idempotent operations:

- `msqueue` [Michael and Scott 1998], `dglmqueue` [Doherty et al. 2004], `folqueue` [Fober et al. 2001] and `rdcss` [Harris et al. 2002] all employ idempotent operations.
- `treiber` [Treiber 1986], `ttaslock` [Herlihy and Shavit 2008, §7.2] and `twalock` [Dice and Kogan 2019] do not employ idempotent operations.

These benchmarks exercise different aspects of internal symmetries so that the individual effects of each symmetry type are more visible.

We also note that we have identified idempotent operations in various widely used concurrency libraries (e.g., `libcds` [Khizhinsky n.d.], `folly` [Facebook n.d.], `ckit` [Bahra n.d.]). Even though Spore's support for C++ precluded us from using `libcds` and `folly` as benchmarks, we did manage to run certain benchmarks from `ckit`, with similar performance gains.

4.3 Results

Our results are summarized in Fig. 3[6]. First, as explained in §1, SR alone is inadequate for scalability, and using a combination of DPOR and SR is crucial: with the exception of a few benchmarks, SR

[6]Detailed tables can be found in [Kokologiannakis et al. 2024b].

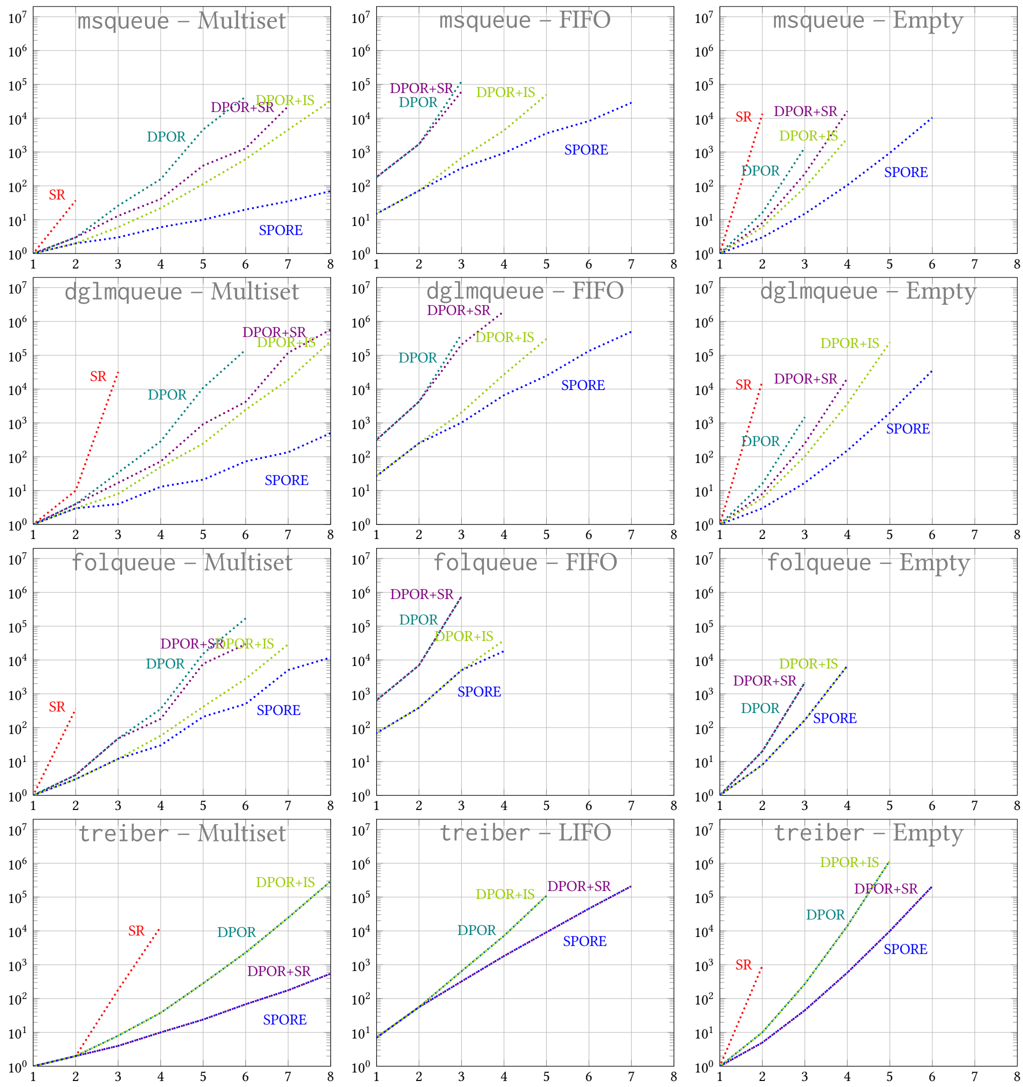


Fig. 3. Data structure benchmarks: Number of executions expored (Y-axis) per input parameter (X-axis)

consistently times out (and we therefore dismiss it for the rest of this discussion). Second, both thread-level and internal symmetries are crucial for scaling to more threads: exclusively either kind of symmetry typically leads to timeouts for some number of threads.

Let us now examine the benchmarks in more detail, starting with the multiset client (left column). The main takeaway from this client is immediately evident: while TruSt typically scales up to 6 threads before timing out, Spore scales gracefully to 8 threads (and more). Looking more closely, however, there are a few other interesting aspects as well.

Starting with `msqueue` and `dglmqueue`[7], TruSt times out for 6 threads and above, while Spore can scale up to many more. The reason for that is simple: the CAS instruction present in the queue's

[7]These benchmarks only differ in their dequeue method, which is why the results are very similar.

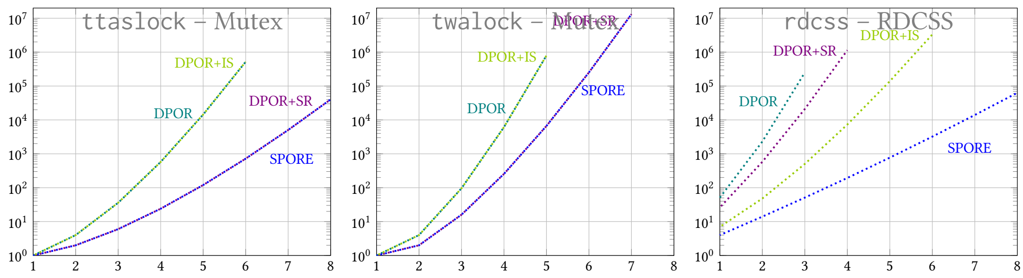


Fig. 4. Non-data-structure benchmarks: Number of executions expored (Y-axis) per input parameter (X-axis)

idempotent operation breaks symmetry, thereby leading to state-space explosion. Spore, on the other hand, runs lickety-split: it explores a single execution when the client is fully symmetric (up to 4 threads), and a small number of executions otherwise (modeling the different ways insertions interfere with deletions). As the number of dequeuers increases, Spore explores more executions,as there are more ways for deletions to interfere with insertions.

Moving on to `folqueue` and `treiber`, we can make observations similar to the ones for the previous benchmarks, albeit a bit toned down. In the case of `folqueue`, thread-level symmetries have a limited effect, as each thread uses a distinct (global) location to dispose pointers, which breaks symmetry among threads early: Spore performs similarly to DPOR+IS, while TruSt performs similarly to DPOR+SR. Analogously, in `treiber`, internal symmetries have no effect, as the code has no idempotent operations: Spore performs just as well as DPOR+SR, while DPOR+IS performs just as well as TruSt.

Generally, we observe that DPOR+IS performs better than DPOR+SR in the multiset client when both thread-level and internal symmetries are present, implying that internal symmetries carry more weight when it comes to scaling to more threads. This should not come as a surprise. Idempotent operations might be performed more than once per thread, while thread-level symmetry will break after the first non-symmetric operation. As such, since the number of idempotent operations is greater than the number of threads, internal symmetries offer a greater reduction.

Next, we move on to the other two clients. In a similar fashion, Spore scales much better than TruSt (which only manages to terminate within the time limit for two or three configurations), although it does not manage to finish within the time limit for all configurations, since these clients are not completely symmetric (like the multiset one). As expected, Spore performs better in the LIFO/FIFO (where it can better leverage the symmetry in the client), and DPOR+IS performs better than DPOR+SR whenever there are internal symmetries, for the same reasons as in the multiset client. (Note that Spore performs similarly to DPOR+IS for the first configuration of each benchmark in the LIFO/FIFO client, as SR requires at least two symmetric threads to have any effect.)

Finally, in Fig. 4 we compare all tools on some non-data-structure benchmarks. The two locking benchmarks do not employ idempotent operations, and thus Spore coincides with DPOR+SR, which has an exponentially smaller state-space than plain DPOR. In contrast, `rdcss` makes heavy use of idempotent operations, and so Spore manages to scale way better than plain DPOR.

5 Related Work

As far as symmetry reduction is concerned, it has mostly been explored in the context of stateful model checking [Clarke et al. 1996; Emerson and Wahl 2005; Wahl and Donaldson 2010]. In that setting, the main challenge is to identify when two threads are symmetric, that is computationally as hard as the graph isomorphism problem. By contrast, Spore is able to detect when two threads are symmetric on-the-fly, though in principle the reductions it achieves are not as good as the ones in stateful model checking.

As far as internal symmetries are concerned, even though a lot of effort has been devoted into making DPOR algorithms more efficient and scalable during the past few years (e.g., [Abdulla et al. 2015, 2017, 2018; Aronis et al. 2018; Chalupa et al. 2017; Chatterjee et al. 2019; Kokologiannakis et al. 2017, 2022, 2019b; Nguyen et al. 2018; Norris and Demsky 2013; Rodríguez et al. 2015]), most works focus on improving the core of DPOR and do not take into consideration the programs under test. SAVer [Kokologiannakis et al. 2021] and LAPOR [Kokologiannakis et al. 2019a] extend DPOR for programs that have spinloops and locks, respectively, while constrained-DPOR [Albert et al. 2018] takes programmer annotations into account in order to consider certain atomic operations non-conflicting.

In a different context, there has been a large body of work on static verification of concurrent programs, with techniques such as bounded model checking (BMC) or abstraction-based techniques (e.g., [Clarke et al. 2004; Elmas et al. 2009; Flanagan et al. 2005; Gavrilenko et al. 2019]). We expect that—at least for SAT/SMT-based techniques—both thread-level and internal symmetries could be exploited in a similar fashion to reduce the size of the resulting SAT formula and speed up the verification.

6 Conclusion

We presented Spore, a novel model checking algorithm that combines DPOR with symmetry reduction, and also exploits internal symmetries of C/C++ concurrent data structures. Our experiments confirm that Spore outperforms the state-of-the-art by a wide margin.

There are several ways this work could be extended. First, we would like to see whether Spore can handle other classes of programs in related domains, namely distributed algorithms and/or persistent programs, where similar symmetries appear. It remains to be seem whether those patterns exhibit symmetries that can be exploited in a similar fashion to enhance the applicability of automated verification techniques in those domains. Second, it would also be interesting whether Spore can be applied to models like ARMv8 [Flur et al. 2016] and POWER [Alglave et al. 2014] that do allow TruSt's $\mathsf{po} \cup \mathtt{rf}$ cycles in consistent executions (which Spore does not currently produce). Finally, Spore could also be combined with testing techniques, so that only representative executions are produced when obtaining traces of a concurrent program.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback. This work was supported by a European Research Council (ERC) Consolidator Grant for the project "PERSIST" under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 101003349).

Data-Availability Statement

The benchmarks and tools used to produce the results of this paper can be found at [Kokologiannakis et al. 2024a]. Spore is available at [Kokologiannakis n.d.].

References

Parosh Aziz Abdulla, Stavros Aronis, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson, and Konstantinos Sagonas. 2015. "Stateless model checking for TSO and PSO." In: *TACAS 2015* (LNCS). Vol. 9035. Springer, Berlin, Heidelberg, 353–367. https://doi.org/10.1007/978-3-662-46681-0_28.

Parosh Aziz Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. 2014. "Optimal dynamic partial order reduction." In: *POPL 2014*. ACM, New York, NY, USA, 373–384. https://doi.org/10.1145/2535838.2535845.

Parosh Aziz Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. Sept. 2017. "Source sets: A foundation for optimal dynamic partial order reduction." *J. ACM*, 64, 4, (Sept. 2017), 25:1–25:49. https://doi.org/10.1145/3073408.

Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, Magnus Lång, Tuan Phong Ngo, and Konstantinos Sagonas. Oct. 10, 2019. "Optimal stateless model checking for reads-from equivalence under sequential consistency." *Proc. ACM Program. Lang.*, 3, (Oct. 10, 2019), 150:1–150:29, OOPSLA, (Oct. 10, 2019). https://doi.org/10.1145/3360576.

Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, and Tuan Phong Ngo. Oct. 2018. "Optimal stateless model checking under the release-acquire semantics." *Proc. ACM Program. Lang.*, 2, OOPSLA, (Oct. 2018), 135:1–135:29. https://doi.org/10.1145/3276505.

Elvira Albert, Miguel Gómez-Zamalloa, Miguel Isabel, and Albert Rubio. 2018. "Constrained dynamic partial order reduction." In: *CAV 2018*. Ed. by Hana Chockler and Georg Weissenbacher. Springer International Publishing, Cham, 392–410. ISBN: 978-3-319-96142-2. https://doi.org/10.1007/978-3-319-96142-2_24.

Jade Alglave, Luc Maranget, and Michael Tautschnig. July 2014. "Herding cats: Modelling, simulation, testing, and data mining for weak memory." *ACM Trans. Program. Lang. Syst.*, 36, 2, (July 2014), 7:1–7:74. https://doi.org/10.1145/2627752.

Stavros Aronis, Bengt Jonsson, Magnus Lång, and Konstantinos Sagonas. 2018. "Optimal dynamic partial order reduction with observers." In: *TACAS 2018* (LNCS). Vol. 10806. Springer, 229–248. https://doi.org/10.1007/978-3-319-89963-3_14.

Samy Al Bahra. N.d. *Concurrency Kit.* (). https://github.com/concurrencykit/ck.

Marek Chalupa, Krishnendu Chatterjee, Andreas Pavlogiannis, Nishant Sinha, and Kapil Vaidya. Dec. 2017. "Data-centric dynamic partial order reduction." *Proc. ACM Program. Lang.*, 2, POPL, (Dec. 2017), 31:1–31:30. https://doi.org/10.1145/3158119.

Krishnendu Chatterjee, Andreas Pavlogiannis, and Viktor Toman. Oct. 2019. "Value-Centric Dynamic Partial Order Reduction." *Proc. ACM Program. Lang.*, 3, OOPSLA, (Oct. 2019). https://doi.org/10.1145/3360550.

Edmund M. Clarke, Somesh Jha, Reinhard Enders, and Thomas Filkorn. 1996. "Exploiting symmetry in temporal logic model checking." *Form. Meth. Syst. Des.*, 9, 1/2, 77–104. https://doi.org/10.1007/BF00625969.

Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. 2004. "A tool for checking ANSI-C programs." In: *TACAS 2004* (LNCS). Vol. 2988. Springer, Berlin, Heidelberg, 168–176. https://doi.org/10.1007/978-3-540-24730-2_15.

Dave Dice and Alex Kogan. 2019. "TWA – Ticket Locks Augmented with a Waiting Array." In: *Euro-Par 2019*. Springer-Verlag, Berlin, Heidelberg, 334–345. ISBN: 978-3-030-29399-4. https://doi.org/10.1007/978-3-030-29400-7_24.

Simon Doherty, Lindsay Groves, Victor Luchangco, and Mark Moir. 2004. "Formal Verification of a Practical Lock-Free Queue Algorithm." In: *FORTE 2004* (LNCS). Ed. by David de Frutos-Escrig and Manuel Núñez. Vol. 3235. Springer, 97–114. https://doi.org/10.1007/978-3-540-30232-2_7.

Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. 2009. "A calculus of atomic actions." In: *POPL 2009*. Ed. by Zhong Shao and Benjamin C. Pierce. ACM, 2–15. https://doi.org/10.1145/1480881.1480885.

E. Allen Emerson and Thomas Wahl. 2005. "Dynamic Symmetry Reduction." In: *TACAS 2005* (LNCS). Ed. by Nicolas Halbwachs and Lenore D. Zuck. Vol. 3440. Springer, 382–396. https://doi.org/10.1007/978-3-540-31980-1_25.

Facebook. N.d. *Folly: Facebook Open-source Library.* (). https://github.com/facebook/folly.

Cormac Flanagan, Stephen N. Freund, and Shaz Qadeer. 2005. "Exploiting Purity for Atomicity." *IEEE Trans. Software Eng.*, 31, 4, 275–291. https://doi.org/10.1109/TSE.2005.47.

Cormac Flanagan and Patrice Godefroid. 2005. "Dynamic partial-order reduction for model checking software." In: *POPL 2005*. ACM, New York, NY, USA, 110–121. https://doi.org/10.1145/1040305.1040315.

Shaked Flur, Kathryn E. Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell. 2016. "Modelling the ARMv8 architecture, operationally: Concurrency and ISA." In: *POPL 2016*. ACM, St. Petersburg, FL, USA, 608–621. ISBN: 978-1-4503-3549-2. https://doi.org/10.1145/2837614.2837615.

Dominique Fober, Yann Orlarey, and Stéphane Letz. 2001. *Optimised Lock-Free FIFO Queue.* Technical Report. GRAME. https://hal.archives-ouvertes.fr/hal-02158792.

Natalia Gavrilenko, Hernán Ponce-de-León, Florian Furbach, Keijo Heljanko, and Roland Meyer. 2019. "BMC for weak memory models: Relation analysis for compact SMT encodings." In: *CAV 2019*. Ed. by Isil Dillig and Serdar Tasiran. Springer International Publishing, Cham, 355–365. ISBN: 978-3-030-25540-4. https://doi.org/10.1007/978-3-030-25540-4_19.

Michalis Kokologiannakis. N.d. *GenMC: Generic model checking for C programs.* (). https://github.com/MPI-SWS/genmc.

Patrice Godefroid. 1997. "Model checking for programming languages using VeriSoft." In: *POPL 1997*. ACM, Paris, France, 174–186. https://doi.org/10.1145/263699.263717.

Timothy L. Harris, Keir Fraser, and Ian A. Pratt. 2002. "A Practical Multi-word Compare-and-Swap Operation." In: *DISC 2002* (LNCS). Ed. by Dahlia Malkhi. Vol. 2508. Springer, 265–279. https://doi.org/10.1007/3-540-36108-1_18.

Maurice Herlihy. 1991. "Wait-Free Synchronization." *ACM Trans. Program. Lang. Syst.*, 13, 1, 124–149.

Maurice Herlihy and Nir Shavit. 2008. *The art of multiprocessor programming.*

Max Khizhinsky. N.d. *CDS C++ library.* (). https://github.com/khizmax/libcds.

Michalis Kokologiannakis, Ori Lahav, Konstantinos Sagonas, and Viktor Vafeiadis. Dec. 2017. "Effective stateless model checking for C/C++ concurrency." *Proc. ACM Program. Lang.*, 2, POPL, (Dec. 2017), 17:1–17:32. https://doi.org/10.1145/3158105.

Michalis Kokologiannakis, Iason Marmanis, Vladimir Gladstein, and Viktor Vafeiadis. Jan. 2022. "Truly stateless, optimal dynamic partial order reduction." *Proc. ACM Program. Lang.*, 6, POPL, (Jan. 2022). https://doi.org/10.1145/3498711.

Michalis Kokologiannakis, Iason Marmanis, and Viktor Vafeiadis. June 2024a. *SPORE: Combining Symmetry and Partial Order Reduction (Replication Package)*. (June 2024). https://doi.org/10.5281/zenodo.10798179.

Michalis Kokologiannakis, Iason Marmanis, and Viktor Vafeiadis. June 2024b. "Spore: Combining Symmetry and Partial Order Reduction (supplementary material)," (June 2024). https://plv.mpi-sws.org/genmc.

Michalis Kokologiannakis, Iason Marmanis, and Viktor Vafeiadis. 2023. "Unblocking Dynamic Partial Order Reduction." In: *CAV 2023*. Vol. 13964. Springer, 230–250. https://doi.org/10.1007/978-3-031-37706-8_12.

Michalis Kokologiannakis, Azalea Raad, and Viktor Vafeiadis. Oct. 2019a. "Effective lock handling in stateless model checking." *Proc. ACM Program. Lang.*, 3, OOPSLA, (Oct. 2019). https://doi.org/10.1145/3360599.

Michalis Kokologiannakis, Azalea Raad, and Viktor Vafeiadis. 2019b. "Model checking for weakly consistent libraries." In: *PLDI 2019*. ACM, New York, NY, USA. https://doi.org/10.1145/3314221.3314609.

Michalis Kokologiannakis, Xiaowei Ren, and Viktor Vafeiadis. 2021. "Dynamic Partial Order Reductions for Spinloops." In: *FMCAD 2021*. IEEE, 163–172. https://doi.org/10.34727/2021/isbn.978-3-85448-046-4_25.

Ori Lahav, Nick Giannarakis, and Viktor Vafeiadis. 2016. "Taming Release-acquire Consistency." In: *POPL 2016*. ACM, St. Petersburg, FL, USA, 649–662. isbn: 978-1-4503-3549-2. https://doi.org/10.1145/2837614.2837643.

Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. 2017. "Repairing sequential consistency in C/C++11." In: *PLDI 2017*. ACM, Barcelona, Spain, 618–632. isbn: 978-1-4503-4988-8. https://doi.org/10.1145/3062341.3062352.

Leslie Lamport. Sept. 1979. "How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs." *IEEE Trans. Computers*, 28, 9, (Sept. 1979), 690–691. https://doi.org/10.1109/TC.1979.1675439.

Maged M. Michael and Michael L. Scott. 1998. "Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors." *J. Parallel Distrib. Comput.*, 51, 1, 1–26.

Huyen T. T. Nguyen, César Rodríguez, Marcelo Sousa, Camille Coti, and Laure Petrucci. 2018. "Quasi-optimal partial order reduction." In: *CAV 2018* (LNCS). Ed. by Hana Chockler and Georg Weissenbacher. Vol. 10982. Springer, 354–371. https://doi.org/10.1007/978-3-319-96142-2_22.

Brian Norris and Brian Demsky. 2013. "CDSChecker: Checking concurrent data structures written with C/C++ atomics." In: *OOPSLA 2013*. ACM, 131–150. https://doi.org/10.1145/2509136.2509514.

César Rodríguez, Marcelo Sousa, Subodh Sharma, and Daniel Kroening. 2015. "Unfolding-based Partial Order Reduction." In: *CONCUR 2015* (LIPIcs). Vol. 42. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 456–469. https://doi.org/10.4230/LIPIcs.CONCUR.2015.456.

SPARC International Inc.. 1994. *The SPARC architecture manual (version 9)*. Prentice-Hall.

R. Kent Treiber. 1986. *Systems Programming: Coping with Parallelism*. Tech. rep. Technical Report RJ5118, IBM. https://dominoweb.draco.res.ibm.com/58319a2ed2b1078985257003004617ef.html.

Thomas Wahl and Alastair Donaldson. 2010. "Replication and Abstraction: Symmetry in Automated Formal Verification." 2, 2, 799–847. https://doi.org/10.3390/sym2020799.

A Formal Model

In this section, we define a simple programming language (§A.1), and show how programs in the language are mapped to execution graphs (§A.2).

A.1 Programming Language

For the purposes of this paper, we introduce a simple imperative programming language with a top-level parallel composition. *Commands*, $c \in \mathsf{Cmd}$, are given by the following grammar:

$$\begin{aligned} c ::= {} & r := e \mid \textbf{error} \mid \textbf{block} \mid \textbf{if } e \textbf{ then } c \mid c_1; c_2 \mid f^{annot}(e_1, \dots, e_n) \\ & \mid r := \textbf{load}^{o,k}(e) \mid \textbf{store}^{o,k}(e_1, e_2) \mid r := \textbf{alloc}() \mid \textbf{free}(e) \end{aligned}$$

where $r \in \mathsf{Reg}$ ranges over *registers*, $f \in \mathsf{Fname}$ over *function names*, $annot \in \mathsf{Annot} \triangleq \{\mathsf{main}, \mathsf{help}\}$ over *function annotations*, $o \in \{\mathsf{na}, \mathtt{rlx}, \mathsf{acq}, \mathtt{rel}, \mathsf{acqrel}, \mathsf{sc}\}$ over RC11 *access modes* [Lahav et al. 2017], $k \subseteq \{\mathsf{excl}\}$ over sets of *attributes*, and $e \in \mathsf{Exp}$ over simple *expressions* built from integers n, registers, and arithmetic operators:

$$e ::= n \mid r \mid e_1 + e_2 \mid e_1 - e_2 \mid \dots$$

RC11 access modes are naturally ordered from weakest to strongest as follows:

$$\begin{tikzcd}
 & & \overset{\text{(acquire)}}{\texttt{acq}} \arrow[dr, phantom, "\sqsubset" sloped] & & \\
\underset{\text{(non-atomic)}}{\texttt{na}} \arrow[r, phantom, "\sqsubset"] & \underset{\text{(relaxed)}}{\texttt{rlx}} \arrow[ur, phantom, "\sqsubset" sloped] \arrow[dr, phantom, "\sqsubset" sloped] & & \texttt{acqrel} \arrow[r, phantom, "\sqsubset"] & \underset{\text{(seq.consistent)}}{\texttt{sc}} \\
 & & \underset{\text{(release)}}{\texttt{rel}} \arrow[ur, phantom, "\sqsubset" sloped] & &
\end{tikzcd}$$

The only attribute we have for accesses is exclusivity flag excl used to denote when an access is part of a read-modify-write (RMW) instruction.

Remark 3. The primitive commands **error** and **block** commands can be used to model `assert` and `assume` statements as follows:

$$\mathtt{assert}(e) \triangleq \textbf{if } \neg e \textbf{ then error} \qquad\qquad \mathtt{assume}(e) \triangleq \textbf{if } \neg e \textbf{ then block}$$

Remark 4. For simplicity, we only allow function calls for the purpose of specifying internal symmetries. Thus, all function calls are annotated and do not return any results.

Remark 5. Commands do not contain any loops, since SMC only works for programs that are guaranteed to terminate. Programs with loops that are guaranteed to terminate in at most k iterations can be rewritten by unfolding those loops k times. Spinloops (even if non-terminating) can soundly be unrolled to a single iteration [Kokologiannakis et al. 2021], e.g., **do** $r := \textbf{load}(x)$ **while**(e) is converted to $r := \textbf{load}(x); \mathtt{assume}(\neg e)$.

A *function definition* has the form $f(r_1, \dots, r_n)$ { **local** $r_{n+1}, \dots, r_{n+m}$; c_{body} }, where f is the function name, $r_1, \dots, r_n$ are the formal parameters, $r_{n+1}, \dots, r_{n+m}$ are the local variables, and c_{body} is the function body. We assume that registers used in c_{body} belong to the function parameters or the declared local variables. For simplicity, we only allow function definitions that satisfy our internal symmetry constraints: the functions do not return any results, and their bodies comprise of a possibly empty sequence of loads followed by a compare-and-swap (CAS) on a location and with an expected value that does not depend on the prior loads, i.e. each c_{body} is of following form:

$$r_1 := \textbf{load}(e_1); \dots; r_k := \textbf{load}(e_k); r_0 := \textbf{load}^{\mathsf{excl}}(e_{loc}); \textbf{if } r_0 = e_{exp} \textbf{ then store}^{\mathsf{excl}}(e_{loc}, e')$$

where $k \geq 0$ and e_{loc} and e_{exp} do not depend on the registers $r_0, \dots, r_k$.

The simplest—and most common—kind of annotateable function is CAS-noret, with performs a CAS without returning whether it was successful.

$$\text{CAS-noret}(\mathit{loc}, \mathit{exp}, \mathit{new})\ \{\ \mathbf{local}\ r;\ \begin{pmatrix} r := \mathbf{load}^{\mathsf{excl}}(\mathit{loc}); \\ \mathbf{if}\ r = \mathit{exp}\ \mathbf{then}\ \mathbf{store}^{\mathsf{excl}}(\mathit{loc}, \mathit{new}) \end{pmatrix}\ \}$$

A program $P \in \mathsf{Prog}$ comprises a sequence of function definitions and a top-level parallel composition of commands $c_1 \| \dots \| c_k$. We assume that all function calls both in the parallel composition and in the function definitions themselves are to previously defined functions (so, in particular, no recursion is allowed) and that the number of arguments matches the number of formal parameters in the function's definition.

A.2 Mapping Programs to Execution Graphs

Commands in our language are interpreted with respect to a *thread identifier* $t \in \mathsf{Tid} \triangleq \mathbb{N}$, a *serial number* $i \in \mathsf{Idx} \triangleq \mathbb{N}$, and an *environment* $\Gamma \in \mathsf{Env} \triangleq (\mathsf{Reg} \rightarrow \mathsf{Val}) \times (\mathsf{Fname} \rightarrow (\mathsf{Reg}^* \times \mathsf{Cmd}))$, mapping registers to values and function names to their definitions. We extend the domain of Γ to expressions e in the expected way; e.g., $\Gamma(e_1 + e_2) = \Gamma(e_1) + \Gamma(e_2)$.

The interpretation of a command $[\![c]\!](t, i, \Gamma)$ yields a set of pairs of the form $\langle o, E\rangle$, where o denotes the command outcome, and $E \in \mathsf{Pexec} \triangleq \mathcal{P}(\mathsf{Event})$ denotes the plain execution graph (i.e., the set of events) leading to o. The outcome o may in turn be either $\bot$ (when the computation has not terminated successfully) or an updated serial number i' along with a new environment Γ'.

The interpretation function $[\![.]\!] : \mathsf{Cmd} \rightarrow (\mathsf{Tid} \times \mathsf{Idx} \times \mathsf{Env}) \rightarrow \mathcal{P}((\mathsf{Idx} \times \mathsf{Env})_\bot \times \mathsf{Pexec})$ for commands is defined by induction over the command syntax.

We start with the base cases, which are straightforward.

$$\begin{aligned}
[\![r := e]\!](t, i, \Gamma) &\triangleq \{\langle (i, \Gamma[r \mapsto \Gamma(e)]), \emptyset\rangle\} \\
[\![\mathbf{block}]\!](t, i, \Gamma) &\triangleq \{\langle \bot, \{\langle t, i, \mathsf{B}\rangle\}\rangle\} \\
[\![\mathbf{error}]\!](t, i, \Gamma) &\triangleq \{\langle \bot, \{\langle t, i, \mathsf{Error}\rangle\}\rangle\} \\
[\![\mathbf{store}^{o,k}(e_1, e_2)]\!](t, i, \Gamma) &\triangleq \{\langle (i+1, \Gamma), \{\langle t, i, \mathsf{W}^{o,k}(\Gamma(e_1), \Gamma(e_2))\rangle\}\rangle\} \\
[\![r := \mathbf{load}^{o,k}(e)]\!](t, i, \Gamma) &\triangleq \left\{\langle (i+1, \Gamma[r \mapsto v]), \{\langle t, i, \mathsf{R}^{o,k}(\Gamma(e), v)\rangle\}\rangle \;\middle|\; v \in \mathsf{Val}\right\} \\
[\![r := \mathbf{alloc}()]\!](t, i, \Gamma) &\triangleq \left\{\langle (i+1, \Gamma[r \mapsto v]), \{\langle t, i, \mathsf{A}(v)\rangle\}\rangle \;\middle|\; v \in \mathsf{Val}\right\} \\
[\![\mathbf{free}(e)]\!](t, i, \Gamma) &\triangleq \{\langle (i+1, \Gamma), \{\langle t, i, \mathsf{D}(\Gamma(e))\rangle\}\rangle\}
\end{aligned}$$

An assignment yields the sigleton set with an updated environment and no events. **block** and **error** yield $\bot$ and a single block event or error event respectively. Stores and deallocations generate a single event, they increment the serial number to account for the generated event, and keep the environment intact. Loads and allocations return a graph with a new event for an arbitrary return value $v \in \mathsf{Val}$ (corresponding to the value read and to the newly allocated memory address respectively), increment the serial number, and return an updated environment with the mapping $r \mapsto v$.

The interpretation of the inductive cases also proceeds as expected, despite it being a bit more complex. The interpretation of **if** e **then** c is determined by the value of $\Gamma(e)$, and either yields the interpretation of c or the empty graph.

$$[\![\mathbf{if}\ e\ \mathbf{then}\ c]\!](t, i, \Gamma) \triangleq \text{if } \Gamma(e) \neq 0 \text{ then } [\![c]\!](t, i, \Gamma) \text{ else } \{\langle (i, \Gamma), \emptyset\rangle\}$$

The interpretation of $c_1; c_2$ comprises two cases, depending on the outcome of c_1: if the computation $[\![c_1]\!](\Gamma)$ terminates successfully, then the resulting outcome is that of c_2 (i.e., o_2), taking the union of the generated executions; otherwise, the interpretation yields $\langle \bot, E_1 \rangle$.

$$\begin{aligned}\llbracket c_1; c_2 \rrbracket(t, i, \Gamma) \triangleq{} & \left\{ \langle o_2, E_1 \cup E_2 \rangle \;\middle|\; \langle (i_1, \Gamma_1), E_1 \rangle \in \llbracket c_1 \rrbracket(t, i, \Gamma) \land \langle o_2, E_2 \rangle \in \llbracket c_2 \rrbracket(t, i_1, \Gamma_1) \right\} \\ & \cup \left\{ \langle \bot, E_1 \rangle \;\middle|\; \langle \bot, E_1 \rangle \in \llbracket c_1 \rrbracket(t, i, \Gamma) \right\}\end{aligned}$$

Finally, function calls are interpreted by first generating a marker event for the function call and then looking up the function definition in the environment Γ, and interpreting its body in a new environment mapping only the parameter names to the values $v_i = \Gamma(e_i)$ passed as arguments.

$$\begin{aligned}\llbracket f^{annot}(e_1, \dots, e_n) \rrbracket(t, i, \Gamma) \triangleq{} & \left\{ \langle (i', \Gamma), \{\langle t, i, \mathsf{M}^{annot}(f, v_1, \dots, v_n) \rangle\} \cup E \rangle \;\middle|\; \langle (i', \_), E \rangle \in \llbracket c \rrbracket(t, i+1, \Gamma') \right\} \\ & \text{where } v_i = \Gamma(e_i) \text{ and } \Gamma(f) = \langle r_1, \dots, r_n, c \rangle \text{ and } \Gamma' = [r_1 \mapsto v_1, \dots, r_n \mapsto e_n]\end{aligned}$$

Observe that the call itself keeps the caller's environment Γ intact, capturing the fact that any updates by the function body are to local variables of the callee and do not propagate to the caller.

Having interpreted commands, we can finally define the interpretation of a program, $\llbracket P \rrbracket$, as a set of execution graphs, whose events come from the interpretations of the individual threads in an environment Γ_0 that maps all the defined function names to their definitions.

$$\left\llbracket \begin{array}{l} f_1(args_1)\ \{\ \texttt{local}\ ls_1;\ body_1\ \} \\ \qquad \vdots \\ f_k(args_k)\ \{\ \texttt{local}\ ls_k;\ body_k\ \} \\ (c_1 \| \dots \| c_n) \end{array} \right\rrbracket \triangleq \begin{array}{l} \text{let } \Gamma_0 = \begin{bmatrix} f_1 \mapsto \langle args_1, body_1 \rangle, \dots, \\ f_k \mapsto \langle args_k, body_k \rangle \end{bmatrix} \text{ in} \\ \left\{ G \in \mathsf{Exec} \;\middle|\; \begin{array}{l} \exists E_1, \dots, E_n. \\ (\forall i.\ (\_, E_i) \in \llbracket c_i \rrbracket(i, 0, \Gamma_0)) \\ \land\ G.\mathsf{E} = \{\mathtt{init}\} \cup E_1 \cup \dots \cup E_n \end{array} \right\} \end{array}$$

We also define an interpretation function $\llbracket . \rrbracket^{\mathsf{Annot}}$, that avoids expanding function calls annotated with help. This is defined exactly as $\llbracket . \rrbracket$ except when applied to function calls annotated with "help", in which case it returns only the function call marker event:

$$\llbracket f^{\mathsf{help}}(e_1, \dots, e_n) \rrbracket^{\mathsf{Annot}}(t, i, \Gamma) \triangleq \{\langle (i+1, \Gamma), \{\langle t, i, \mathsf{M}^{\mathsf{help}}(f, v_1, \dots, v_n) \rangle\} \rangle\}$$

In contrast, function calls annotated with "main" are properly expanded:

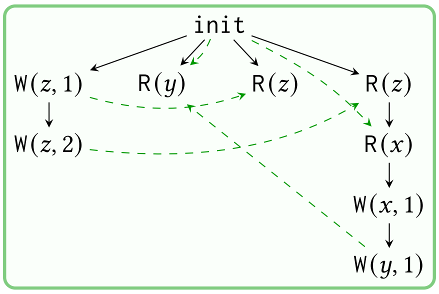
$$\begin{aligned}& \llbracket f^{\mathsf{main}}(e_1, \dots, e_n) \rrbracket^{\mathsf{Annot}}(t, i, \Gamma) \triangleq \\ & \qquad \left\{ \langle (i', \Gamma), \{\langle t, i, \mathsf{M}^{\mathsf{main}}(f, v_1, \dots, v_n) \rangle\} \cup E \rangle \;\middle|\; \langle (i', \_), E \rangle \in \llbracket c \rrbracket^{\mathsf{Annot}}(t, i+1, \Gamma') \right\} \\ & \text{where } v_i = \Gamma(e_i) \text{ and } \Gamma(f) = \langle r_1, \dots, r_n, c \rangle \text{ and } \Gamma' = [r_1 \mapsto v_1, \dots, r_n \mapsto e_n]\end{aligned}$$

B Changing Maximal Extensions for Symmetry Reduction

Modifying the revisiting condition of DPOR fails to restore completeness for Symmetry Reduction. To see why, consider the following program

```
                  ||            || T3: r := z                ||  T4: r := z
                  ||            ||       if (r ≠ 1)          ||        if (r ≠ 1)
T1: z := 1        || T2: a := y ||         b := CAS(x, 0, 1) ||          b := CAS(x, 0, 1)     (rmw-extens)
    z := 2        ||            ||         if (b = 1)        ||          if (b = 1)
                  ||            ||           y := 1          ||            y := 1
```

where T3 and T4 are symmetric. The following execution is a consistent execution of rmw-extens that satisfies our criterion for breaking symmetries (irreflexive(symb; eco)).



We argue that this execution cannot be generated without changing the set of affected events by a revisit (i.e., by simply restricting the graph w.r.t. the porf-prefix of the revisitor).

To reach this execution, the write to y of T4 must revisit the read of T2, and thus the execution before the revisit must have T3 be maximally-extended with respect to the rest of the execution. After the read of z of T3 reads the maximal value (1), the succeeding RMW operation has no other consistent option but to read from the write to y of T4 (due to the atomicity constraint on RMW operations), but this forces the eco order to contradict the thread order. Therefore, the maximal-extension definition for this read cannot allow us to obtain the execution in question.

One possible solution to this problem might be to change the maximality condition of the reads that are symb-before some event in the po-prefix of the revisiting write, but this does not work for two reasons.

First, in case there are multiple such reads (e.g., if the symmetric threads of rmw-extens also read a bunch of other variables before and after reading z, whose values are not used), which one should obey the non-standard maximality condition? How can we pick the "right" read?

Second, even if we somehow select the "right" read to change its maximality condition—or, say, we change the maximality condition for all reads that have the option of consistently reading from a co-earlier write—then reading from which write should be deemed the "maximal" one? Suppose that we add many other writes $z := 3$ to T1 of rmw-extens before the $z := 2$ and both before and after $z := 1$. How would we pick the only "right" write event (namely, W$(x, 1)$) to deem it as the "maximal" one?

C Helping Sufficient Conditions

C.1 Source Event

We present an example showing the importance of the source event being observed by the main thread. In the following program, both CASes can succeed by reading from the $x := 1$ of T1, which is, however, not observed before the main CAS.

```
T1: $r := y$   ||  T2: $\boxed{\mathsf{CAS}^{\mathsf{main}}(x, 1, 2)}$  ||  T3: $\boxed{\mathsf{CAS}^{\mathsf{help}}(x, 1, 2)}$
    $x := 1$   ||      $y := 1$                                 ||
```

(NO-SOURCE-READ)

This creates a problem because T1 cannot read $r = 1$ when the main CAS succeeds (due to the porf-cycle); yet T1 can read $r = 1$ when the helping CAS succeeds, as shown in the execution graphs below.

$$\begin{tikzcd}
 & \mathtt{init} \arrow[dl] \arrow[d] \arrow[dr] \arrow[dd, dashed, bend left] & \\
\mathsf{R}(y) \arrow[d] & \mathsf{M}^{\mathsf{main}}(\ldots) \arrow[d] & \mathsf{M}^{\mathsf{help}}(\ldots) \arrow[d] \\
\mathsf{W}(x, 1) \arrow[rr, dashed, bend left] & \mathsf{R}^{\mathsf{excl}}(x) \arrow[dd] & \mathsf{R}^{\mathsf{excl}}(x) \arrow[d] \\
 & & \mathsf{W}^{\mathsf{excl}}(x, 2) \\
 & \mathsf{W}(y, 1) \arrow[uuul, dashed, bend left] &
\end{tikzcd}
\qquad
\begin{tikzcd}
 & \mathtt{init} \arrow[dl] \arrow[d] \arrow[dr] & \\
\mathsf{R}(y) \arrow[d] & \mathsf{M}^{\mathsf{main}}(\ldots) \arrow[d] & \mathsf{M}^{\mathsf{help}}(\ldots) \\
\mathsf{W}(x, 1) \arrow[r, dashed] & \mathsf{R}^{\mathsf{excl}}(x) \arrow[d] & \\
 & \mathsf{W}^{\mathsf{excl}}(x, 1) \arrow[d] & \\
 & \mathsf{W}(y, 1) \arrow[uuul, dashed, bend left] &
\end{tikzcd}$$

Consequently, one cannot soundly eliminate the helping CAS in this case, and so we impose the condition that the source write must be observed by a thread before performing an operation that might be helped by other threads.

C.2 Linearization Events

We present an example justifying why we need the immediate predecessor of the main function f_m to be observed before the helping function f_h. In the following program, this condition does not hold, and indeed the depicted execution, which reveals an error, would not be detected if we treated the helping function as a no-op. Our condition would avoid such scenarios by enforcing that, if there is no annotation error, any behavior in the main function happens independently (in a porf sense) of the helping function since execution are porf acyclic.

```
T1: $r := x$                                  ||  T2: $r := x$
    if $(r = 0)$                              ||      if $(r = 0)$
        $\boxed{\mathsf{CAS}^{\mathsf{main}}(x, 0, 1)}$  ||          $\boxed{\mathsf{CAS}^{\mathsf{help}}(x, 0, 1)}$
    else                                      ||
        assert(false)                         ||
```

(LIN-EVENTS)

$$\begin{tikzcd}
 & \mathtt{init} \arrow[dl] \arrow[dr] \arrow[dr, dashed, bend left] & \\
\mathsf{R}(x) & & \mathsf{R}(x) \arrow[d] \\
 & & \mathsf{M}^{\mathsf{help}}(\ldots) \arrow[d] \\
 & & \mathsf{R}^{\mathsf{excl}}(x) \arrow[d] \\
 & & \mathsf{W}(x, 1) \arrow[uuull, dashed]
\end{tikzcd}$$

D Completeness of Internal Symmetries

In this section, we instantiate the memory model M to RC11 [Lahav et al. 2017] with minor modifications. Below we provide the memory-model definition, as well as the corresponding annotation error definition.

D.1 Memory Model

Definition D.1 (Consistent Execution). Given an execution G, $\text{consistent}_{\mathsf{M}}(G)$ is the conjunction of the following predicates: (1) porf is irreflexive, (2) $\mathsf{hb};\mathsf{eco}$ is irreflexive, (3) $\mathsf{rmw} \cap (\mathsf{rb};\mathsf{eco}) = \emptyset$, and (4) psc is acyclic, where

- $\mathsf{sw} \triangleq [\mathsf{E}^{\sqsupseteq\mathsf{rel}}];([\mathsf{F}];\mathsf{po})?;[\mathsf{W}^{\sqsupseteq\mathsf{rlx}}];(\mathsf{rf};\mathsf{rmw})^*;\mathsf{rf};[\mathsf{R}^{\sqsupseteq\mathsf{rlx}}];(\mathsf{po};[\mathsf{F}])?;[\mathsf{E}^{\sqsupseteq\mathsf{acq}}]$
- $\mathsf{hb} \triangleq (\mathsf{po} \cup \mathsf{sw})^+$
- $\mathsf{psc} \triangleq [\mathsf{E}^{\mathsf{sc}}];([\mathsf{F}];\mathsf{hb})?;(\mathsf{hb}|_{\mathsf{loc}} \cup \mathsf{co} \cup \mathsf{rb} \cup \mathsf{po} \cup \mathsf{po};\mathsf{hb};\mathsf{po});(\mathsf{hb};[\mathsf{F}])?;[\mathsf{E}^{\mathsf{sc}}] \cup [\mathsf{F}^{\mathsf{sc}}];\mathsf{hb};\mathsf{eco};\mathsf{hb};[\mathsf{F}^{\mathsf{sc}}]$

Definition D.2 (Erroneous Execution). Given an execution G, $\textsc{IsErroneous}_{\mathsf{M}}(G)$ is the disjunction of the following predicates:

- G contains an explicit error event: $G.\mathtt{Error} \neq \emptyset$.
- G contains a memory access or deallocation event e to an unallocated memory address, i.e., where G either does not contain a $\mathtt{A}(\mathtt{loc}(e))$ event, or where such an event exists but is not hb-before e.
- G contains a *use-after-free* error, i.e., a deallocation event that is not hb-after all accesses to the deallocated location.
- G contains more than one deallocation events to the same location.
- G contains a *data race*, i.e., two hb-unordered accesses to the same location, at least one of which is a write, and at least one of which is "non-atomic".

We lift $\mathtt{mod}(f_m)$ to return the successful access mode of the CAS instruction of $f_m \in \mathsf{M}^{\mathsf{main}}$.

D.2 Annotation Error

Definition D.3. An event $f_h \in \mathsf{M}^{\mathsf{help}}$ in an execution G is *matched*, if there exist (a) a corresponding $f_m \in G.\mathsf{M}^{\mathsf{main}}$ with $\mathtt{arg}(f_m) = \mathtt{arg}(f_h)$ and (b) a source write $s \in G.\mathsf{W}$ with $\mathtt{loc}(s) = \mathtt{loc}(f_m)$ and $\mathtt{val}(s) = \mathtt{exp}(f_m)$ (c) events $l_w \in G.\mathsf{W}^{\sqsupseteq\mathsf{rel}}$ and $l_r \in G.\mathsf{R}^{\sqsupseteq\mathsf{acq}}$ such that:

- $\langle s, f_m\rangle \in G.\mathsf{rf}^?;[s'];\mathsf{po}$
- $\langle f_m, f_h\rangle \in \mathsf{po}^{-1}|_{\mathrm{imm}};[l_w];G.\mathsf{rf};[l_r];\mathsf{po}$
- $\mathtt{mod}(l_r) \not\sqsubset \mathtt{mod}(l_w)$

Definition D.4 (Annotation error). An execution G is *correctly annotated* if every $f_h \in G.\mathsf{M}^{\mathsf{help}}$ is matched by an $f_m \in G.\mathsf{M}^{\mathsf{main}}$ with a source write s such that for all $w \in \mathit{rng}([s];\mathsf{co})$, $w \in \mathsf{W}^{\mathsf{excl}}$ and $\mathtt{val}(w) \neq \mathtt{val}(s)$.

D.3 Program Assumptions

We assume that every annotated function consists of an optional sequence of reads followed by a CAS instruction. We also assume that, if an annotated function includes a read instruction, then the successful access mode of the CAS instruction is $\sqsupseteq \mathtt{acqrel}$, and the failing access mode is $\sqsupseteq \mathtt{acq}$. Otherwise, i.e., if there are no read instructions in the annotated function, we require that the failed access mode of the CAS is at least as strong as the successful access mode of the CAS's read. The reads have no external dependencies, apart from a possible data dependency to the write of the CAS instruction.

D.4 Completeness

The following lemma shows that we can remove the events of a helping function whose CAS instruction failed, while preserving both consistency and the presence of an error.

Lemma D.5. *Let G be a consistent and erroneous execution and $f_h \in \mathsf{M}^{\mathsf{help}}$ a function call whose CAS instruction fails. Then, the execution that results from removing the events of f is also consistent and erroneous.*

Proof. Since the CAS fails, all removed events are reads. Read events can only contribute to errors related to the lack of hb synchronization. Such errors, however, are also checked using the f_h event. Consistency is preserved by removing events. □

The following lemma presents the conditions under which we can replace the implementation of a main function (whose CAS instruction fails) with the implementation of a helping function whose CAS instruction succeeds and remove the helping function implementation events, while preserving consistency and the presence of an error.

Lemma D.6. *Let G be a consistent and erroneous execution and $f_h \in \mathsf{M}^{\mathsf{help}}$, such that G contains f_h's implementation events. If f_h's CAS r_h succeeds, f_h is matched by $f_m \in \mathsf{M}^{\mathsf{main}}$ with a source event s, r_h reads from s, and f_m is either* po*-maximal or followed by a failed CAS r_m s.t. $\langle r_h, r_m\rangle \in G.\mathsf{rmw}; (G.\mathsf{rf}; G.\mathsf{rmw})^*; G.\mathsf{rf}$, then the execution where f_m's implementation is replaced by f_h's implementation, and f_h's implementation is removed, is consistent and erroneous.*

Proof. Let w_h be the CAS write of f_h. Since f_h is matched by f_m, there is a linearization write l_w $\mathsf{po}|_{\mathrm{imm}}$-before f_m and a linearization read po-before f_h, such that $\langle l_w, l_r\rangle \in G.\mathsf{hb}$. We reason about the resulting execution, by interpreting the transformation as if we replace the events of f_m with a dummy event, and swap the incoming and outgoing po edges of f_m's and f_h's events with each other. We write e_m to refer to r_m and the corresponding CAS write in G'. The helping CAS events (r_h and the following write) in G become e_m in G'.

By construction, the only possible additional edges in the resulting execution G' are po from and to events of $G'.f_m$ (which were previously events of f_h). Any $G'.\mathsf{po}$ edge to an event m of f_m (from an event not in f_m) also exists as a $\mathsf{po}?; [l_w]; \mathsf{hb}; [l_r]; \mathsf{po}; [m]$ in G. Any $G'.\mathsf{po}$ edge from an event m of $G'.f_m$ (to an event not in f_m, in which case f_m is not po-maximal) also exists as a $[m]; \mathsf{po}?; [r_h]; (\mathsf{rf}; \mathsf{rmw})^*; \mathsf{rf}; [r_m]; \mathsf{po}$ in G. Therefore, porf acyclicity is preserved. If the annotated functions contain read instructions, the aforementioned path from m is in hb (both CASes are at least `acqrel`), and therefore any additional po edge in G' is an $\mathsf{hb}; \mathsf{po}$ edge in G. Otherwise, G' has additional hb edges that start with $[e_m]; \mathsf{po}$ (in which case f_m is not po-maximal in G). Any hb edge that does not start with e_m also exists in G, since an po to e_m in G' is an $\mathsf{po}?; \mathsf{hb}; \mathsf{po}$ in G and an sw to e_m in G' is an $\mathsf{sw}; (\mathsf{rf}; \mathsf{rmw})^*; \mathsf{rf} \subseteq \mathsf{sw}$ in G.

Any hb-unordered error, i.e., error due to a lack of hb ordering between two events, is preserved since only hb edges originating from e_m are added in G'. To see this, consider such an error between two (different) events a and b of G. If none of the events are the helping CAS, then the events remain hb-unordered in G' (only hb edges from e_m, which is equivalent to the helping CAS in G, are added). Otherwise, if one of them is the helping CAS, then the error remains since events in $\mathsf{M}^{\mathsf{help}}$ are explicitly checked for hb-unordered errors. Additionally, the transformation does not remove any write events, and it therefore preserves possible annotation errors.

To see that $G'.\mathsf{psc}$ is acyclic, we will show that any psc cycle would also exist in G. First, we consider additional psc edges due to the added po that ends in e_m. We have shown that any $\mathsf{po}; [e_m]$ edge in G' is an $\mathsf{po}?; \mathsf{hb}; \mathsf{po}; [e_m]$ edge in G. The only interesting case is when the $\mathsf{po}; [e_m]$ edge is $\mathsf{po}|_{\mathrm{imm}}$, i.e., starts with l_w (in every other case, replacing po with $\mathsf{po}; \mathsf{hb}; \mathsf{po}$ preserves the psc). In this case, however, the $[\mathsf{E^{sc}}]; ([\mathsf{F}]; \mathsf{hb})?; [l_w]; \mathsf{po}|_{\mathrm{imm}}; [e_m]; \ldots$ edge in G' is in psc^2 in G, using the $[\mathsf{W^{sc}}]; \mathsf{sw}; [\mathsf{R^{sc}}]$ edge from l_w to l_r. Therefore, no psc edge can be added due to the $\mathsf{po}; [e_m]$ edge.

Second, we consider which additional psc edges can exist due to the added po edges that start from e_m in G' (in which case f_m is not po-maximal in G). Observe that the edges in $\mathsf{hb}|_{\mathsf{loc}}$ that start with po and cannot be written as $\mathsf{po} \cup \mathsf{po}; \mathsf{hb}; \mathsf{po}$ are of the form $(\mathsf{po}; \mathsf{hb}?; \mathsf{sw})|_{\mathsf{loc}}$. The only additional psc edge are thus in one of the following forms

(1) $[\mathsf{E^{sc}}]; ([\mathsf{F}]; \mathsf{hb})?; (\mathsf{po}; \mathsf{hb}; [e_m]; \mathsf{po}); (\mathsf{hb}; [\mathsf{F}])?; [\mathsf{E^{sc}}]$
(2) $[\mathsf{E^{sc}}]; ([\mathsf{F}]; \mathsf{hb})?; ((\mathsf{po}; \mathsf{hb}?; \mathsf{sw})|_{\mathsf{loc}} \cup \mathsf{po} \cup \mathsf{po}; \mathsf{hb}; \mathsf{po}); [e_m]; (\mathsf{po}; \mathsf{hb}?; [\mathsf{F}]); [\mathsf{E^{sc}}]$
(3) $[\mathsf{E^{sc}}]; ([\mathsf{F}]; \mathsf{hb})?; [e_m]; ((\mathsf{po}; \mathsf{hb}?; \mathsf{sw})|_{\mathsf{loc}} \cup \mathsf{po} \cup \mathsf{po}; \mathsf{hb}; \mathsf{po}); (\mathsf{hb}; [\mathsf{F}])?; [\mathsf{E^{sc}}]$

It is easy to see that in the first two cases, the same edge exists in G: any $[e_m]; \mathsf{po}$ part of the edge is preceded by a $\mathsf{po} \cup \mathsf{hb} \cup \mathsf{sw}$ edge, and thus the same path exists via the r_m event in G.

In the last case, if the psc edge starts with $[\mathsf{F}]; \mathsf{hb}; [e_m]$, then it also exists in G for the same reason (there is an hb to r_m in G). Therefore, only psc edges of the form $[e_m]; ((\mathsf{po}; \mathsf{hb}?; \mathsf{sw})|_{\mathsf{loc}} \cup \mathsf{po} \cup \mathsf{po}; \mathsf{hb}; \mathsf{po}); (\mathsf{hb}; [\mathsf{F}])?; [\mathsf{E^{sc}}]$, can be added. In any psc cycle in G', the $\mathsf{psc}; [e_m]$ edge ends in $\mathsf{sw} \cup \mathsf{po} \cup \mathsf{co} \cup \mathsf{rb}$. In every case, the same edge exists to r_m, and therefore the same cycle appears in G (any po edge from e_m in G' is a po edge from r_m in G). □

Our completeness theorem states that if a there is a consistent and erroneous execution under the original semantics, then there is a consistent and erroneous execution under the annotated semantics.

THEOREM D.7. *Let $G \in [\![P]\!]_M$ such that IsERRONEOUS$_M(G)$. Then, there exists $G' \in [\![P]\!]_M^{\mathsf{Annot}}$ such that IsERRONEOUS$_{SYM}(G)$.*

PROOF. Let G be an erroneous execution of $[\![P]\!]_{\mathsf{M}}$ with the minimal number of helping function implementation events (such an execution exists from the hypothesis). If G contains no events corresponding to the implementation of a helping function, then the result follows immediately ($G \in [\![P]\!]_{\mathsf{M}}^{\mathsf{Annot}}$). Otherwise, we will show that there exists a (not necessarily full) consistent and erroneous execution with at least one less helping function implementation. We can then maximally extend this execution (without adding implementation events of helping functions), and obtain a contradiction from the assumption about G.

Let $f_h \in G.\mathsf{M^{help}}$ that has implementation events. If the CAS of f_h is failing, from Lemma D.5 we obtain a full, consistent, and erroneous execution without f_h's implementation events. Otherwise, the CAS of f_h succeeds. If f_h is not matched, we restrict G to the $\mathsf{porf}?$-prefix of f_h. The resulting execution is erroneous and contains fewer helping function implementation events, concluding this case.

Otherwise, the CAS of f_h succeeds and f_h is matched by a $f_m \in \mathsf{M^{main}}$. Let s_m be the source event of f_m and r_h the read of f_h's CAS. If r_h is reading from $w_d \neq s_m$, then $\langle s_m, w_d \rangle \in G.\mathsf{co}$ due to coherence (since $\langle s_m, f_h \rangle \in G.\mathsf{hb}$). Restrict G to the porf-prefix of r_h. The resulting execution does not contain r_h (since G is porf-acyclic). Therefore we can remove the rest of f_h's implementation events and obtain an execution with fewer helping function implementation events that is also erroneous: there is a write $w_d \in rng([s_m]; \mathsf{co})$ that writes the expected value of f_h.

Otherwise, the CAS of f_h succeeds, f_h is matched by $f_m \in \mathsf{M^{main}}$, and r_h is reading from s_m. If f_m's CAS succeeds, the two CAS events cannot be reading from the same write (due to atomicity). So, f_m must be reading from some write w_1 such that $\langle s_m, w_1 \rangle \in G.\mathsf{co}$ (due to coherence, since $\langle s_m, f_m \rangle \in G.\mathsf{rf}^?; \mathsf{po}$). Let w_m be the CAS write event of f_m's implementation, w_h the CAS write event of f_h's implementation, and let G' be the restriction of G to the set $dom(G.\mathsf{porf}?; [\{w_m, w_h\}])$. It is easy to see that in G', at least one of w_m, w_h are porf-maximal (because G is porf-acyclic), and there are at least two writes that write the expected value to the CAS location (the writes w_1

and s_m that f_m's and f_h's CAS are reading, respectively). We consider two cases, depending on which one is porf-maximal. If w_h is porf-maximal we remove f_h's implementation events. If w_m is porf-maximal we remove f_m's implementation events and apply Lemma D.6 (f_m is po-maximal). In both cases, the resulting execution G'' has fewer helping function implementation events and is erroneous: $w_1, s_m, f_h \in G''.\mathsf{E}$.

Otherwise, the CAS of f_h succeeds, f_h is matched by $f_m \in \mathsf{M}^{\mathsf{main}}$, r_h is reading from s_m, and f_m's CAS fails. Let r_m the CAS read of f_m, w_h the CAS write of f_h. If $\langle w_h, r_m\rangle \in (G.\mathsf{rf}; G.\mathsf{rmw})^+; G.\mathsf{rf}$, we get a consistent and erroneous execution without f_h's implementation events from Lemma D.6. Otherwise, there is a write event co-after w_h and $\mathsf{co}?;\mathsf{rf}$-before r_m that is not part of an RMW. Let w' be the co-latest such event. It is $\langle w', r_m\rangle \in G.\mathsf{porf}$, and therefore $\langle f_m, w'\rangle \notin G.\mathsf{porf}$. If $\langle f_h, w'\rangle \in G.\mathsf{porf}$, we restrict G to the $\mathsf{porf}?$-prefix of w' and f_m, in which f_m is porf-maximal, and apply Lemma D.6. Otherwise, we restrict to the $\mathsf{porf}?$-prefix of w' and f'_h, in which f_h is porf-maximal. In both cases, the resulting execution has fewer helping function implementation events and is erroneous (contains f_h and w'), concluding the final case of our proof. $\square$

E Algorithm Correctness

In this section, we prove the correctness and optimality of our algorithm. The correctness is independent of the chosen memory model M, provided it satisfies some assumptions (§E.2).

In the sequel, we assume a program P s.t. $[\![[]\!]_{\mathsf{M}}]G$ only contains executions of finite size. Since $\mathrm{consistent}_{\mathsf{SYM}}(G)$ is more restrictive than $\mathrm{consistent}_{\mathsf{M}}(G)$, the same holds for $[\![[]\!]_{\mathsf{SYM}}]G$. We omit mentioning the program and the semantics when there is no ambiguity.

E.1 Definitions

E.1.1 Preliminaries.

Definition E.1 (Available). The available set $avail(G)$ of an execution G is the set of events that can extend the execution under the program semantics.

Definition E.2 (Full Execution). An execution G is full if it has no available events.

$$full(G) \triangleq avail(G) = \emptyset$$

Definition E.3 (Open Read). An event $r \in \mathsf{R}$ of an execution G is *open* if it s part of an RMW operation and there exists a matching write event in the available set of G.

$$open(G, r) \triangleq \exists w \in avail(G) \cap \mathsf{W}.\ \langle r, w \rangle \in \mathsf{rmw}$$

In the sequel, we lift $G.\mathsf{symb}$ to also include the available events of the execution G.

E.1.2 Operational steps.

Operational steps $G \overset{e\,@e'}{\longrightarrow} G'$ capture the non-revisit steps of the algorithm, namely that a graph G is extended by adding an extra event e that is $G'.\mathsf{cb}_{\mathsf{SYM}}$-maximal in the resulting graph G'. The event e' (if applicable) contains the extra information necessary to add e in a unique fashion (i.e., its rf-predecessor in the case of reads, and its co-predecessor in the case of writes).

READ
$$\frac{r \in avail(G) \cap \mathsf{R} \qquad p \in G.\mathsf{W} \qquad dom(G.\mathsf{symb}; [r]) \in G.\mathsf{E}}{G \overset{r\,@p}{\longrightarrow} \mathrm{AddRead}(G, r, p)}$$

WRITE
$$\frac{w \in avail(G) \cap \mathsf{W} \qquad p \in G.\mathsf{W} \qquad dom(G.\mathsf{symb}; [w]) \in G.\mathsf{E}}{G \overset{w\,@p}{\longrightarrow} \mathrm{AddWrite}(G, w, p)}$$

Definition E.4 (Prefix). Given executions G and G', G is a *prefix* of G', if G can be extended to G' with a series of operational steps.

$$G \sqsubseteq G' \triangleq G \longrightarrow^{*} G'$$

E.1.3 Maximal Steps.

Definition E.5 (Maximal). An event $w \in \mathsf{W}$ of an execution G is *maximal* if it has no co successor.

$$\textsc{IsMaximal}(G, w) \triangleq rng([w]; G.\mathsf{co}) = \emptyset$$

Maximal steps are those steps that add an event in co-maximal fashion, and thus allow the event to be affected by a revisit.

Read

$$\frac{r \in \mathsf{R} \qquad G \overset{r@p}{\rightarrow} G' \qquad \textsc{IsMaximal}(G,p)}{G \overset{r}{\rightsquigarrow} G'}$$

Write

$$\frac{\nexists r \in G.\mathsf{R}.\ open(G,r) \qquad G \overset{w@p}{\rightarrow} G' \qquad \textsc{IsMaximal}(G,p)}{G \overset{w}{\rightsquigarrow} G'}$$

Write-Exclusive

$$\frac{w \in \mathsf{W} \qquad r \in \mathsf{R} \qquad \langle r,w\rangle \in \mathsf{rmw} \qquad \bar{G} \overset{r}{\rightsquigarrow} G \overset{w@p}{\rightarrow} G' \qquad \textsc{IsMaximal}(G,p)}{G \overset{w}{\rightsquigarrow} G'}$$

Definition E.6 (Maximal Completion). Given an execution G and an event e, $\textsc{MaxCompletion}(G,e)$ is the unique execution G' such that $G \overset{e}{\rightsquigarrow} G'$, if such an execution exists, and $\bot$ otherwise.

We lift $\textsc{MaxCompletion}(G,\cdot)$ to sequences of events, with $\textsc{MaxCompletion}(\bot,\_) = \bot$.

E.2 Memory-Model

We assume an underlying memory model M with $\mathsf{cb}_\mathsf{M} = (\mathsf{po} \cup \mathsf{rf})^+$, whose consistency predicate $\mathsf{consistent}_\mathsf{M}(\cdot)$ satisfies the following axioms.

Axiom 1 (cb_M Acyclicity). *Given an execution G, if* $\mathsf{consistent}_M(G)$*, then* $\textsc{acyclic}(G.\mathsf{cb}_M)$.

Axiom 2 (Prefix Closedness). *Given an execution G and a set $E \subseteq G.\mathsf{E}$ such that* $\mathsf{consistent}_M(G)$ *and $dom(G.\mathsf{cb}_M; [E]) \subseteq E$, it is* $\mathsf{consistent}_M(G|_E)$.

Axiom 3 (Extensibility). *Given an execution G, events $e \in avail(G), w \in G.\mathsf{W}$, and an execution G' such that $G \overset{e@w}{\rightarrow} G'$, $\textsc{IsMaximal}(G,w)$, and $\forall r \in G.\mathsf{R}.\ open(G,r) \Rightarrow \textsc{IsMaximal}(G, G.\mathsf{rf}(r))$, if* $\mathsf{consistent}_M(G)$*, then* $\mathsf{consistent}_M(G')$.

We then define a new memory model SYM, with

$$\mathsf{cb}_\mathsf{SYM} \triangleq (\mathsf{po} \cup \mathsf{rf} \cup \mathsf{symb})^+$$
$$\mathsf{consistent}_\mathsf{SYM}(G) \triangleq \mathsf{consistent}_\mathsf{M}(G) \land \textsc{irreflexive}(\mathsf{symb};\mathsf{eco})$$

E.3 Algorithm

Definition E.7 (Configuration). A *configuration* is a tuple $\langle G, <\rangle$, where G is an execution and $<$ is a total order on $G.\mathsf{E}$.

Definition E.8 (Subsequence). Given two sequences a and b, $a \sqsubseteq b$ stands for a being a subsequence of b.

We lift the notion of a prefix to configurations pointwise, i.e., $\langle G, <\rangle \sqsubseteq \langle G', <'\rangle$ if $G \sqsubseteq G'$ and $< \sqsubseteq <'$, where the total orders are interpreted as the respective sequences.

Algorithmic steps $\langle G, <\rangle \Rightarrow \langle G', <'\rangle$ occur whenever $\textsc{Explore}(G)$ calls $\textsc{Explore}(G')$ recursively: they capture the conditions that $\textsc{Explore}(G)$ checks and the computation it performs to calculate a G' in order to invoke itself recursively on G'.

Non-revisit Step

$$\frac{e \in avail(G) \qquad G \overset{e@p}{\rightarrow} G' \qquad \mathsf{consistent}_{\mathsf{SYM}}(G')}{G \underset{nr}{\overset{e@p}{\Rightarrow}} G'}$$

Write Revisit Step

$$\frac{\begin{array}{c} w \in avail(G) \cap \mathsf{W} \qquad r \in G.\mathsf{R} \\ P \triangleq dom((G \uplus \{w\}).\mathsf{cb}_{\mathsf{SYM}}; [\{w\}]) \qquad r \notin P \\ [a_1, \cdots, a_n] = \mathsf{sort}_<(\{e \in G.\mathsf{E} \setminus P \mid r \le e\}) \\ G'' \overset{a_1}{\rightsquigarrow} \cdots \overset{a_n}{\rightsquigarrow} G \qquad G'' \overset{w@p}{\rightarrow} \overset{r@w}{\rightarrow} G' \qquad \mathsf{consistent}_{\mathsf{SYM}}(G') \end{array}}{\langle G, < \rangle \underset{rv\ r}{\overset{w@p}{\Rightarrow}} \langle G', \mathsf{Restrict}(<, G''.\mathsf{E} \cup \{r\}) \mathbin{+\!\!+} [w] \rangle}$$

Definition E.9 (Production Sequence). A production sequence S is a sequence of algorithm steps that start from $\langle G_\emptyset, \emptyset \rangle$ and Apply(S) is the unique configuration obtained by applying all the steps of S.

We abuse notation and also write Apply(S) for the projected execution of the configuration.

E.4 Lemmas and Propositions

Corollary E.10 (Unique Extensibility). *Given two execution such that $G_\emptyset \sqsubseteq G \rightsquigarrow G'$, if* $\mathsf{consistent}_{SYM}(G)$*, then* $\mathsf{consistent}_{SYM}(G')$.

Proof. From $\mathsf{consistent}_{\mathsf{SYM}}(G)$, we have $\mathsf{consistent}_{\mathsf{M}}(G)$ and irreflexive$(G.\mathsf{symb}; G.\mathsf{eco})$. From Axiom 3, it is $\mathsf{consistent}_{\mathsf{M}}(G)$. Since $G_\emptyset \sqsubseteq G$, it is $dom(G.\mathsf{symb}; [G.\mathsf{E}]) \subseteq G.\mathsf{E}$. Let e be the added event. By definition of $\rightsquigarrow$, $rng([e]; G'.\mathsf{eco}) = \emptyset$. Thus, any $G'.\mathsf{symb}; G'.\mathsf{eco}$ loop must have a $G'.\mathsf{symb}$ edge that starts from e, contradicting $dom(G.\mathsf{symb}; [G.\mathsf{E}]) \subseteq G.\mathsf{E}$. Therefore, $\mathsf{consistent}_{\mathsf{SYM}}(G')$. □

Corollary E.11 (Prefix Closedness). *Given two executions G and G' s.t. $G \sqsubseteq G'$, if* $\mathsf{consistent}_{SYM}(G')$*, then* $\mathsf{consistent}_{SYM}(G)$.

Proof. From Axiom 2 and the fact that irreflexivity is prefix-closed. □

Proposition E.12. *For any execution graph G, $G_\emptyset \sqsubseteq G$ iff* acyclic$(G.\mathsf{cb}_{SYM})$ *and* $dom(G.\mathsf{symb}; [G.\mathsf{E}]) \subseteq G.\mathsf{E}$.

Proposition E.13. *For any execution graph G s.t. $G_\emptyset \Rightarrow^* G$, $G_\emptyset \sqsubseteq G$.*

Proof. Proof by induction on the length of the production sequence. The non-revisit step case is trivial. For the revisit step $G \underset{rv\ r}{\overset{w@-}{\Rightarrow}} G'$, we have $dom(G.\mathsf{symb}; [w]) \subseteq G.\mathsf{E}$. For the execution G'' after the removal of the affected events, we have $G_\emptyset \sqsubseteq G''$, since $G'' \rightsquigarrow^+ G$ (i.e., , the maximal steps in $G_\emptyset \sqsubseteq G$ can commute to the end). Therefore, from the revisit condition $G'' \rightarrow\rightarrow G'$, we have $G_\emptyset \sqsubseteq G'$.

□

Proposition E.14. *For any execution graph G s.t. $G_\emptyset \Rightarrow^* G$, $dom(G.\mathsf{symb}; [\mathsf{next}_P(G)]) \subseteq G.\mathsf{E}$.*

Proof. $\mathsf{next}_P(\cdot)$ always picks the left-most available event e, and therefore there can be no event in $avail(G)$ that is symb-before e. □

Proposition E.15. *For any execution graph G s.t. $G_\emptyset \Rightarrow^* G$,* acyclic$(G.\mathsf{cb}_{SYM})$ *and* $dom(G.\mathsf{cb}_{SYM}; [G.\mathsf{E}]) \subseteq G.\mathsf{E}$.

Proof. Follows from Prop. E.13 and Prop. E.12. ◻

Definition E.16. Given a production sequence S that ends in G, i.e., $G = \textsc{Apply}(S)$, we define $rv(S)$ as the subset of events that revisited in a revisit step of S, i.e., $rv(S) \triangleq \left\{ w \in \mathsf{W} \;\middle|\; \exists \overset{w@\_}{\underset{rv\ \_}{\Rightarrow}} \in S \right\}$, and $mrv(S)$ the subset of $rv(S)$ such that the respective revisited read event was not later revisited or deleted in S, i.e.,

$$mrv(S) \triangleq \left\{ w \in \mathsf{W} \;\middle|\; \begin{array}{l} \exists S_r, t \in \overset{w@\_}{\underset{rv\ r}{\Rightarrow}}.\, S_r +\!\!+\, t \sqsubseteq S \;\wedge \forall S', t'. \\ S_r +\!\!+\, t \sqsubset S' +\!\!+\, t' \sqsubseteq S \Rightarrow t' \notin \overset{-}{\underset{rv\ r}{\Rightarrow}} \wedge\; r \in \textsc{Apply}(S' +\!\!+\, t').\mathsf{E} \end{array} \right\}$$

Lemma E.17. *Let S be a production sequence s.t. $\langle G, < \rangle = \textsc{Apply}(S)$. If $\langle G, < \rangle \overset{w@\_}{\underset{rv\ r}{\Rightarrow}} \langle G', <' \rangle$, then $[G.\mathsf{E}]; G.\mathsf{cb}_{SYM}?; [G'.\mathsf{E} \setminus \{r\}] \subseteq [G'.\mathsf{E}]; G'.\mathsf{cb}_{SYM}?; [G'.\mathsf{E}]$.*

Proof. By definition of the revisit step, all $\mathsf{po} \cup \mathsf{rf} \cup \mathsf{symb}$ edges ending in events that are not affected by the revisit, i.e., the revisited read and the deleted events, remain unaltered. ◻

Lemma E.18. *Let S be a production sequence s.t. $\langle G, < \rangle = \textsc{Apply}(S)$. If $\langle G, < \rangle \overset{w@\_}{\underset{rv\ \_}{\Rightarrow}} \langle G', <' \rangle$, then $mrv(S) \cap G.\mathsf{E} \subseteq G'.\mathsf{E}$.*

Proof. Let $w' \in mrv(S) \cap G.\mathsf{E}$ and assume that $w' \notin G'.\mathsf{E}$. Since w' revisited a read r that was not revisited or deleted later in S, it must be $\langle w', r \rangle \in G.\mathsf{rf}$ and $w' > r$. For the revisit step from G to G', we have that there is an execution G'' such that $G'' \rightsquigarrow^* G_1 \overset{w'}{\rightsquigarrow} G_2 \rightsquigarrow^* G$. Because $w' > r$, it is $\langle w', r \rangle \in G_2.\mathsf{rf}$, which contradicts $G_1 \overset{w'}{\rightsquigarrow} G_2$. ◻

Lemma E.19. *Let S be a production sequence s.t. $\langle G, < \rangle = \textsc{Apply}(S)$.
Then, $rv(S) \subseteq dom([G.\mathsf{E}]; G.\mathsf{cb}_{SYM}?; [mrv(S)])$.*

Proof. Proof by induction on the length of the production sequence. For the empty sequence the result is trivial ($rv(\emptyset) = \emptyset$). Let S be a production sequence, and $S' = S +\!\!+\, t$, i.e., $\langle G, < \rangle \Rightarrow \langle G', <' \rangle$ where $\langle G, < \rangle = \textsc{Apply}(S)$ and $\langle G', <' \rangle = \textsc{Apply}(S')$. If t is a non-revisit step then the result is obvious since $mrv(S) = mrv(S')$, $rv(S) = rv(S')$, and $G.\mathsf{cb}_{\mathsf{SYM}} \subseteq G'.\mathsf{cb}_{\mathsf{SYM}}$. Otherwise, t is a $\overset{w''@\_}{\underset{rv\ \_}{\Rightarrow}}$ step.

Let $w \in rv(S')$. If $w = w''$, is is obvious that $w \in dom([G'.\mathsf{E}]; G'.\mathsf{cb}_{\mathsf{SYM}}?; [mrv(S')])$ ($w'' \in mrv(S')$). Otherwise, $w \in rv(S)$ ($rv(S') = rv(S) \cup \{w''\}$). From the inductive hypothesis, $rv(S) \subseteq dom([G.\mathsf{E}]; G.\mathsf{cb}_{\mathsf{SYM}}?; [mrv(S)])$. From Lemma E.18 and Lemma E.17,
$[G.\mathsf{E}]; G.\mathsf{cb}_{\mathsf{SYM}}?; [mrv(S) \cap G.\mathsf{E}] \subseteq [G'.\mathsf{E}]; G'.\mathsf{cb}_{\mathsf{SYM}}?; [G'.\mathsf{E}]$. Since $mrv(S) \subseteq rv(S)$ and $rv(S) \subseteq G.\mathsf{E}$, it is $mrv(S) \subseteq G.\mathsf{E}$ and it suffices to show that $mrv(S) \subseteq dom(G'.\mathsf{cb}_{\mathsf{SYM}}?; [mrv(S')])$. Let $w' \in mrv(S)$ and r the respective event that w' revisited. It is $r < w'$ and $\langle w', r \rangle \in G.\mathsf{rf}$. If r was not revisited or deleted by t, our result follows immediately ($w' \in mrv(S')$). Otherwise, t revisited r' and revisited or deleted $r \geq r'$. From Lemma E.18, $w' \in G'.\mathsf{E}$, and since $w' > r'$, it must be $\langle w', w'' \rangle \in G'.\mathsf{cb}_{\mathsf{SYM}}$, concluding our proof ($w'' \in mrv(S')$). ◻

Proposition E.20. *Let $\langle G, < \rangle$ be a configuration s.t. $\langle G_\emptyset, \emptyset \rangle \Rightarrow^* \overset{w@-}{\underset{rv\ -}{\Rightarrow}} \langle G, < \rangle$. There is no production sequence starting from $\langle G, < \rangle$ that deletes w.*

Proof. Follows directly from Lemma E.19 since in any later production sequence S, $w \in rv(S)$. ◻

Proposition E.21 (Termination). *Any production sequence has finite length.*

Proof. From Prop. E.20, any event that revisited cannot be deleted. Therefore, the lexicographical order $\langle rv(S), \textsc{Apply}(S).\mathsf{E}\rangle$ increases with each algorithm step, and is bounded above since $rv(S) \subseteq \textsc{Apply}(S).\mathsf{E}$ and the program semantics only contain executions of finite size. □

Proposition E.22 (Well-formedness). *Given an execution G such that $G_\emptyset \Rightarrow^+ G$, G has at most one open read, and if it has an open read $r \in G.\mathsf{R}$, then r was added or revisited in the last step.*

Proof. Proof by induction on the length of the production sequence S. The base case where there is only one step in S is trivial (there can only be one event e that was added in the previous step). Let two executions such that $G \Rightarrow G'$, and t the corresponding step. If $\mathsf{next}_P(G) \notin \mathsf{W}$, t is a non-revisit step and the result follows immediately: only $\mathsf{next}_P(G)$ can be open in G', since G has at most one open read from the inductive hypothesis, which $\mathsf{next}_P(G)$ would pick. If t is a non-revisit step, G' has no open reads. Otherwise, t is a revisit step. Let r' the revisited read, $\bar{G}$ the last execution in S before G where r' was added or last revisited. Additionally, let $\bar{S} \sqsubseteq S$ the production sequence of $\bar{G}$. All events of $\bar{G}$ are either $\bar{\lesssim}$-before r' or in the $\mathsf{cb}_{\mathsf{SYM}}$?-prefix of an event in $mrv(\bar{S})$ (the write that revisited r' to reach $\bar{G}$). From Prop. E.20, the latter events cannot be deleted. Additionally, no other event of $\bar{G}$ can be revisited before t. Assume $r'' \in \bar{G}.\mathsf{E}$ is revisited by a write w''. It is $r'' \bar{\lesssim} r'$, and since r' is not revisited or deleted until t, r' is in the $\mathsf{cb}_{\mathsf{SYM}}$?-prefix of w''. From Prop. E.20, this contradicts that t revisits r': it would delete w'' (Lemma E.17), but w'' revisited earlier an event.

Since no event of $\bar{G}$ is revisited until t, and only the events of $\bar{G}$ that are $\bar{\lesssim}$-before r' can be deleted, it is $\bar{G} \setminus \{r'\} \sqsubseteq G'$. Additionally, all events in $G'.\mathsf{E} \setminus \bar{G}.\mathsf{E}$ are in the $\mathsf{cb}_{\mathsf{SYM}}$?-prefix of a write, and therefore cannot contain an open read. Therefore, the only open read in G' can be r', which was just revisited. □

Proposition E.23. *Let two production sequences S and S' with $S \sqsubset S'$, and let $\langle G, <\rangle \triangleq \textsc{Apply}(S)$ and $\langle G', <'\rangle \triangleq \textsc{Apply}(S')$. Then, $G \sqsubseteq G'$ iff no step of $S' \setminus S$ revisits an event of G. Additionally, if no step of $S' \setminus S$ revisits an event of G, then $< +\!\!+ \mathsf{next}_P(G) \sqsubseteq <'$.*

Proof. We will first show the forward direction. Assume there is a step in $S' \setminus S$ that revisits an event of G and let r be a $<$-minimal event such event of $G.\mathsf{E}$, and t the first revisit of r in $S' \setminus S$. Let $\langle G_1, <_1\rangle$ and $\langle G_2, <_2\rangle$ the configurations before and after t, respectively, and w' be the event that revisits in t. It is easy to see that $G.\mathsf{rf}(r) = G_1.\mathsf{rf}(r)$, and therefore $w \triangleq G.\mathsf{rf}(r) \neq G_2.\mathsf{rf}(r) = w'$. From the revisit condition, it must be $w <_1 r$ or $\langle w, w'\rangle \in (G_1 \uplus \{w'\}).\mathsf{cb}_{\mathsf{SYM}}$ (and therefore $w \in G_2.\mathsf{E}$): in the other case, w would be in the deleted set and the maximality condition would fail. In either case, from Prop. E.20, no step of S' after G_2 can delete w. Therefore, it cannot be that r reads from w in G', contradicting $G \sqsubseteq G'$.

For the opposite direction, it is easy to see that if no event of G is revisited, it must be $\langle G, <\rangle \sqsubset \langle G', <'\rangle$, and since $\mathsf{next}_P(G)$ will be added in the first step of $S' \setminus S$ and no earlier event is revisited, it is $< +\!\!+ \mathsf{next}_P(G) \sqsubseteq <'$. □

Lemma E.24 (Prefix Extension). *Given two executions G and G' s.t. $G \sqsubset G'$, and an event $w \in G'.\mathsf{E} \setminus G.\mathsf{E}$, there is a unique execution G_p s.t. $G \sqsubseteq G_p \sqsubseteq G'$ and $G_p.\mathsf{E} \setminus G.\mathsf{E} = dom(G'.\mathsf{cb}_{\mathsf{SYM}}; [\{w\}]) \setminus G.\mathsf{E}$. We call this execution the* prefix-extension *of G until w of G', and denote it as $\textsc{PrefixExtension}(G, G', w)$.*

Lemma E.25 (Next Prefix). *Given two executions G_s and G_t s.t. $G_s \sqsubset G_t$, and an event $e \in avail(G_s) \cap G_t.\mathsf{E}$, if there is no step s.t. $G_s \overset{e@-}{\rightarrow} G' \sqsubseteq G_t$, then $e \in \mathsf{R}$, $w \triangleq G_t.\mathsf{rf}(e) \notin G_s.\mathsf{W}$, and $G_p \overset{w@p}{\rightarrow} \overset{e@w}{\rightarrow} G_c \sqsubseteq G_t$, where $G_p \triangleq \textsc{PrefixExtension}(G_s, G_t, w)$ and $p \triangleq G_t|_{G_p.\mathsf{E} \cup \{w\}}.\mathsf{co}(w)$ Additionally, for any execution $G' \sqsupset G_s$ such that $e \in G'.\mathsf{E}$, it is $G_c \sqsubseteq G'$.*

Algorithm 2 Production sequence

```
procedure ProductionSequence($G_f$)
    $S \leftarrow \emptyset$
    while Apply($S$) $\neq G_f$ do
        $S \leftarrow$ GetNext($S, G_f$)
    return $S$
procedure GetNext($S_0, G_t$)
    $G_{S_0} \leftarrow$ Apply($S_0$)
    $e_0 \leftarrow \mathsf{next}_P(G_{S_0})$
    if $\exists p. G_{S_0} \xrightarrow{e_0 @ p} G' \land G' \sqsubseteq G_t$ then
        return $S_0 \mathbin{++} \underset{nr}{\overset{e_0 @ p}{\Rightarrow}}$
    $w \leftarrow G_t.\mathsf{rf}(e_0)$
    $G_p \leftarrow$ PrefixExtension($G_{S_0}, G_t, w$)
    $p \leftarrow G_t|_{G_p.\mathsf{E} \cup \{w\}}.\mathsf{co}(w)$
    $\langle S, A \rangle \leftarrow \langle S_0, \emptyset \rangle$
    while true do
        $e \leftarrow \mathsf{next}_P$(Apply($S$))
        if $e = w$ then
            return $S \mathbin{++} \underset{rv\ e_0}{\overset{w @ p}{\Rightarrow}}$
        if $e \notin G_p.\mathsf{E}$ then
            $A \leftarrow A \mathbin{++} e$
        $S \leftarrow$ GetNext($S$, MaxCompletion($G_p, A$))
```

E.5 Completeness and Optimality

Lemma E.26. *Let S_0 be a production sequence, and G_t an execution such that* $\mathsf{consistent}_{SYM}(G_t)$, *Apply($S$) $\triangleq G_{S_0}$, $G_\emptyset \sqsubseteq G_{S_0} \sqsubset G_t$ and* $\mathsf{next}_P(G_{S_0}) \in G_t.\mathsf{E}$. *Then GetNext($S_0, G_t$) returns a production sequence $S' \sqsupset S$ such that $G_{S_0} \sqsubset G' \sqsubseteq G_t$, where $G' =$ Apply(S'). Additionally, for any production sequence $\hat{S} \sqsupset S_0$ such that Apply($\hat{S}$) $\sqsupseteq G_t$, it is $\hat{S} \sqsupseteq S'$.*

Proof sketch: By induction on the lexicographical order $\langle -|G_{S_0}.\mathsf{E}|, |G_t.\mathsf{E}| \rangle$ of the arguments. Since the program semantics only contain executions of finite size and $G_{S_0} \sqsubset G_t$, the measure is bounded below.

Let: $\langle G_{S_0}, <_{S_0} \rangle \triangleq$ Apply(S_0)

Assume: 1. $G_\emptyset \sqsubseteq G_{S_0} \sqsubset G_t$
2. $\mathsf{consistent}_{\mathsf{SYM}}(G_t)$
3. $\mathsf{next}_P(G_{S_0}) \in G_t.\mathsf{E}$

Let: $S' \triangleq$ GetNext(S_0, G_t)

Prove: 1. $S_0 \sqsubset S'$, Apply(S') $= \langle G', <' \rangle$, and $G_{S_0} \sqsubset G' \sqsubseteq G_t$
2. $<_{S_0} \mathbin{++} \mathsf{next}_P(G_{S_0}) \sqsubseteq <'$, last($S'$) either adds or revisits $\mathsf{next}_P(G_{S_0})$
3. If $\hat{S} \sqsupset S_0$ and Apply($\hat{S}$) $\sqsupseteq G_t$, then $\hat{S} \sqsupseteq S'$

⟨1⟩1. Suffices: Prove ⟨0⟩ under the assumption that it holds for any pair $\langle S_0', G_t' \rangle$ such that $\langle -|$Apply$(S_0').\mathsf{E}|, |G_t'.\mathsf{E}| \rangle < M \triangleq \langle -|G_{S_0}.\mathsf{E}|, |G_t.\mathsf{E}| \rangle$.

PROOF: Induction on the lexicographical order $\langle -|\textsc{Apply}(S_0).\mathsf{E}|, |G_t.\mathsf{E}|\rangle$ of the arguments $\langle S_0, G_t\rangle$. The order is bounded below because the program semantics only contain executions with finite size and $G_{S_0} \sqsubset G_t$.

$\langle 1\rangle 2.$ CASE: The test in Line 9 succeeds.

$\langle 2\rangle 1.$ Proof obligations 1 and 2 of $\langle 0\rangle$hold.

PROOF: From §E.3 and $\text{consistent}_{\text{SYM}}(G')$ ($\text{consistent}_{\text{SYM}}(G_t)$ and Corollary E.11).

$\langle 2\rangle 2.$ Proof obligation 3 holds.

$\langle 3\rangle 1.$ $p \in G_{S_0}.\mathsf{E} \cup \{\_\}$

PROOF: From $\langle 1\rangle 2$.

LET: Let t the step in Line 9

$\langle 3\rangle 2.$ ASSUME: $\hat{S} \sqsupseteq S +\!\!+\, t'$, with $t \neq t'$

PROVE: FALSE

$\langle 4\rangle 1.$ $S \sqsubset \hat{S}$ and $\textsc{Apply}(S) \sqsubseteq \textsc{Apply}(\hat{S})$

$\langle 4\rangle 2.$ $\hat{S} \setminus S$ does not revisit an event of $G_{S_0}.\mathsf{E}$ and $e_0 \in \textsc{Apply}(\hat{S}).\mathsf{E}$.

PROOF: From $\langle 4\rangle 1$ and Prop. E.23.

$\langle 4\rangle 3.$ Q.E.D.

PROOF: Contradiction from $\langle 4\rangle 2$, $\langle 3\rangle 1$ and $t \neq t'$: $p \in G_{S_0}$ and it will not be deleted later, therefore e_0 will always be in the wrong placement p' of t'.

$\langle 2\rangle 3.$ Q.E.D.

$\langle 1\rangle 3.$ CASE: The test in Line 9 fails.

$\langle 2\rangle 1.$ $e_0 \in \mathsf{R}$ and $G_t.\mathtt{rf}(e_0) \notin G_{S_0}.\mathtt{W}$

PROOF: From Lemma E.25.

$\langle 2\rangle 2.$ LET: 1. $w \triangleq G_t.\mathtt{rf}(e_0)$

2. $G_p \triangleq \textsc{PrefixExtension}(G_{S_0}, G_t, w)$

3. $p \triangleq G_t|_{G_p.\mathsf{E}\cup\{w\}}.\mathtt{co}(w)$

$\langle 2\rangle 3.$ $\text{consistent}_{\text{SYM}}(G_p)$

PROOF: From $\text{consistent}_{\text{SYM}}(G_t)$, $G_p \sqsubseteq G_t$, and Corollary E.11.

$\langle 2\rangle 4.$ $\textsc{acyclic}(G_t.\mathtt{cb}_{\text{SYM}})$

PROOF: From $G_\emptyset \sqsubseteq G_t$ and Prop. E.12.

$\langle 2\rangle 5.$ ASSUME: $e_0 \in G_p.\mathsf{E}$

PROVE: FALSE

$\langle 3\rangle 1.$ $\langle e_0, w\rangle \in G_t.\mathtt{cb}_{\text{SYM}}$

PROOF: From $\langle 2\rangle 5$, the definition of G_p, and $e_0 \notin G_{S_0}.\mathsf{E}$.

$\langle 3\rangle 2.$ $\langle w, e_0\rangle \in G_t.\mathtt{rf}$

PROOF: By definition of w.

$\langle 3\rangle 3.$ Q.E.D.

PROOF: Contradiction from $\langle 3\rangle 1$, $\langle 3\rangle 2$, and $\langle 2\rangle 4$.

$\langle 2\rangle 6.$ ASSUME: $\hat{S} \sqsupset S_0$ and $\textsc{Apply}(\hat{S}) \sqsupseteq G_t$

PROVE: There exist $\bar{S}$, $\bar{G}$, and $\bar{A}$ such that

1. $p \triangleq G_t|_{G_p.\mathsf{E}\cup\{w\}}.\mathtt{co}(w)$

2. $S_0 \sqsubset \bar{S} +\!\!+ \underset{rv\ e_0}{\overset{w@p}{\Rightarrow}} \sqsubseteq \hat{S}$

3. $\textsc{Apply}(S_0) \sqsubset \textsc{Apply}(\bar{S}) = \langle \bar{G}, \bar{<}\rangle$

4. $\bar{G} = \textsc{MaxCompletion}(G_p, \bar{A})$

5. $\bar{A} = \text{sort}_{\bar{<}}(\bar{G}.\mathsf{E} \setminus G_p.\mathsf{E})$

6. $\text{next}_P(\bar{G}) = w$

$\langle 3\rangle 1.$ There is no step in $\hat{S} \setminus S_0$ that revisits an event of G.

PROOF: From Prop. E.23.

⟨3⟩2. Every configuration $\langle G', <' \rangle$ in $\hat{S} \setminus S_0$ after the first, it is $G_{S_0} \sqsubset G'$, $e_0 \in G'.\mathsf{E}$, and $<_{S_0} \mathbin{++} e_0 \sqsubseteq <'$.

PROOF: From Prop. E.23 and $\mathsf{next}_P(G_{S_0}) = e_0$.

⟨3⟩3. LET: Let t be the last step that revisits e_0 in $\hat{S}$, and $\langle \bar{G}, \bar{<} \rangle$ and $\langle G'', <'' \rangle$ the configurations that precede and follow t, respectively.

PROOF: There exists a step that revisits e_0 in $\hat{S}$ from ⟨3⟩1, ⟨3⟩2, and ⟨1⟩3.

⟨3⟩4. $G' \sqsupseteq G_c$, where $G_p \overset{w@p}{\rightarrow} \overset{e_0@w}{\rightarrow} G_c$

PROOF: From ⟨3⟩2 and Lemma E.25.

⟨3⟩5. Q.E.D.

PROOF: From ⟨3⟩4 and ⟨3⟩2, w must revisit e_0 from an execution $\bar{G}$. Then, $\bar{S}$ is the part of $\hat{S}$ up to $\bar{G}$, $\bar{A}$ the events affected by the revisit, and the rest proof obligations follow immediately by the definition of the revisit step and ⟨3⟩2.

⟨2⟩7. LET: Given an iteration with values $\langle S, A \rangle$,
$\langle G(S), <_S \rangle \triangleq \text{APPLY}(S)$ and $G_{MC}(A) \triangleq \text{MAXCOMPLETION}(G_p, A)$

⟨2⟩8. LET: $\mathsf{Inv}(S, A)$ be the conjunction of
1. $G_{MC}(A) \neq \bot$
2. $S = S_0 \Rightarrow A = \emptyset$
3. $S \neq S_0 \Rightarrow S_0 \sqsubset S \land G_{S_0} \sqsubset G(S) \sqsubseteq G_{MC}(A) \land <_{S_0} \mathbin{++} e_0 \sqsubseteq <_S$
4. $A = \mathsf{sort}_{<_S}(G(S).\mathsf{E} \setminus G_p.\mathsf{E})$ and $w \notin A$
5. If $\exists e_o \in A.\ \mathit{open}(G(S), e_o)$, then $e_o = \text{LAST}(A)$
6. If $\hat{S} \sqsupset S_0$ and $\text{APPLY}(\bar{S}) \sqsupseteq G_t$, then $\bar{A} \sqsupseteq A$ and $\bar{S} \sqsupseteq S$ (⟨2⟩6)

⟨2⟩9. At the beginning of an iteration $\langle S, A \rangle$: $\mathsf{Inv}(S, A)$

⟨3⟩1. $\mathsf{Inv}(S_0, \emptyset)$

PROOF: It is $G(S_0) = G_{S_0} \sqsubseteq G_p = G_{MC}(\emptyset) \neq \bot$. The rest follow trivially from $S = S_0$ and $A = \emptyset$.

⟨3⟩2. SUFFICES ASSUME: An iteration $\langle S, A \rangle$ where $\mathsf{Inv}(S, A)$ and the test in Line 17 fails
PROVE: It is $\mathsf{Inv}(S', A')$ for the values $\langle S', A' \rangle$ at the end of the iteration.

PROOF: ⟨3⟩1 and induction on the number of loop iterations until the test in Line 17 succeeds.

⟨3⟩3. $G_{S_0} \sqsubseteq G(S) \sqsubseteq G_{MC}(A)$ and $<_{S_0} \sqsubseteq <_S$

⟨4⟩1. $G_{S_0} \sqsubseteq G(S_0) \sqsubseteq G_{MC}(\emptyset)$

PROOF: From $G_{S_0} = G(S_0)$, $G_{MC}(\emptyset) = G_p$, and $G_{S_0} \sqsubseteq G_p$ (definition of G_p).

⟨4⟩2. Q.E.D.

PROOF: From loop invariants 3, 2, and ⟨4⟩1.

⟨3⟩4. $e \neq \bot$

⟨4⟩1. $\mathit{avail}(G_{MC}(A)) \neq \emptyset$

PROOF: From $w \in \mathit{avail}(G_p)$ and $w \notin A$ (4), we have $w \in \mathit{avail}(G_{MC}(A))$.

⟨4⟩2. $G(S) \sqsubseteq G_{MC}(A)$

PROOF: From ⟨3⟩3.

⟨4⟩3. Q.E.D.

PROOF: From ⟨4⟩1 and ⟨4⟩2.

⟨3⟩5. CASE: $e \in G_p.\mathsf{E}$

⟨4⟩1. $A' = A$

⟨4⟩2. $G_{MC}(A') \neq \bot$

PROOF: From loop invariant 1 and ⟨4⟩1.

⟨4⟩3. $\mathsf{consistent}_{\mathsf{SYM}}(G_{MC}(A'))$

PROOF: From ⟨2⟩3 and Corollary E.10.

⟨4⟩4. $\mathsf{next}_P(G(S)) \in G_{MC}(A')$
Proof: From $e \in G_p.\mathsf{E} \subseteq G_{MC}(A).\mathsf{E} = G_{MC}(A').\mathsf{E}$ (⟨4⟩1).
⟨4⟩5. $G(S) \sqsubset G_{MC}(A')$
⟨5⟩1. $G(S) \sqsubseteq G_{MC}(A')$
Proof: From ⟨3⟩3 and ⟨4⟩1.
⟨5⟩2. $G(S) \neq G_{MC}(A')$
Proof: From ⟨4⟩4.
⟨5⟩3. Q.E.D.
⟨4⟩6. $\langle -|\textsc{Apply}(S).\mathsf{E}|, |G_{MC}(A').\mathsf{E}|\rangle < M$
⟨5⟩1. $G(S) \sqsupseteq G_{S_0}$
Proof: From ⟨3⟩3.
⟨5⟩2. Case: $G(S) \neq G_{S_0}$
Proof: From ⟨5⟩1 and ⟨5⟩2, it is $G(S) \sqsubset G_{S_0}$ and therefore $-|G(S).\mathsf{E}| < -|G_{S_0}|.\mathsf{E}$.
⟨5⟩3. Case: $G(S) = G_{S_0}$
⟨6⟩1. $\mathsf{next}_P(G(S)) = e_0$
Proof: From ⟨5⟩3 and loop invariant 3 it is $S = S_0$.
⟨6⟩2. Q.E.D.
Proof: Contradiction from ⟨6⟩1, ⟨3⟩5, and ⟨2⟩5.
⟨5⟩4. Q.E.D.
Proof: Cases ⟨5⟩2 and ⟨5⟩3 are exhaustive.
⟨4⟩7. 1. $S_0 \sqsubset S'$, $\textsc{Apply}(S') = \langle G(S'), <_{S'}\rangle$, and $G(S) \sqsubset G(S') \sqsubseteq G_{MC}(A')$
2. $<_S \mathbin{+\!\!+} \mathsf{next}_P(G(S)) \sqsubseteq <_{S'}$ and $\textsc{last}(S')$ either adds or revisits $\mathsf{next}_P(G(S))$
3. If $\bar{S} \sqsupset S_0$ and $\textsc{Apply}(\bar{S}) \sqsupseteq G_{MC}(A')$, then $\bar{S} \sqsupseteq S'$
Proof: From ⟨4⟩5, ⟨4⟩3, ⟨4⟩4, ⟨4⟩6, and ⟨1⟩1.
⟨4⟩8. $S' \neq S_0$
Proof: From ⟨4⟩7:1.
⟨4⟩9. $S_0 \sqsubset S' \wedge G_{S_0} \sqsubset G(S') \sqsubseteq G_{MC}(A') \wedge <_{S_0} \mathbin{+\!\!+} e_0 \sqsubseteq <'_S$
Proof: From ⟨4⟩7 and ⟨3⟩3.
⟨4⟩10. $A' = \mathsf{sort}_{<_{S'}}(G(S').\mathsf{E} \setminus G_p.\mathsf{E})$ and $w \notin A'$
⟨5⟩1. $A = \mathsf{sort}_{<_S}(G(S).\mathsf{E} \setminus G_p.\mathsf{E})$ and $w \notin A$
Proof: From loop invariant 4.
⟨5⟩2. $G(S').\mathsf{E} \setminus G(S).\mathsf{E} \subseteq G_p.\mathsf{E}$
From ⟨5⟩1, it is $A' = A \subseteq G(S).\mathsf{E}$, and from ⟨4⟩7 (1) it is $G(S) \sqsubset G(S') \sqsubseteq G_{MC}(A')$.
⟨5⟩3. $G(S').\mathsf{E} \setminus G_p.\mathsf{E} = G(S).\mathsf{E} \setminus G_p.\mathsf{E}$
Proof: From ⟨5⟩2.
⟨5⟩4. Q.E.D.
Proof: From ⟨5⟩3, $<_S \sqsubseteq <_{S'}$ (⟨4⟩7:2), ⟨5⟩1, and ⟨4⟩1.
⟨4⟩11. If $\exists e_o \in A'.\, \mathit{open}(G(S'), e_o)$, then $\Rightarrow e_o = \textsc{last}(A')$
⟨5⟩1. If $\exists e_o \in G(S).\, \mathit{open}(G(S'), e_0)$, then $\mathit{open}(G(S), e_0)$
Proof: From $G(S) \sqsubseteq G(S')$ (⟨4⟩7:1).
⟨5⟩2. Q.E.D.
Proof: From loop invariant 5, ⟨5⟩1, and ⟨4⟩1.
⟨4⟩12. If $\hat{S} \sqsupset S_0$ and $\textsc{Apply}(\bar{S}) \sqsupseteq G_t$, then $\bar{A} \sqsupseteq A'$ and $\bar{S} \sqsupseteq S'$
⟨5⟩1. Assume: $\hat{S} \sqsupset S_0$ and $\textsc{Apply}(\bar{S}) \sqsupseteq G_t$
⟨5⟩2. $\bar{A} \sqsupseteq A$ and $\bar{S} \sqsupseteq S$
Proof: From loop invariant 6 and ⟨2⟩6 (2).
⟨5⟩3. $G_{MC}(A') \sqsubseteq \bar{G}$

Proof: $G_{MC}(A') = G_{MC}(A) \sqsubseteq G_{MC}(\bar{A}) = \bar{G}$ ($A \sqsubseteq \bar{A}$ from ⟨5⟩2).

⟨5⟩4. $\bar{S} \sqsupseteq S'$

Proof: From ⟨4⟩7(3), ⟨5⟩1, and ⟨5⟩3.

⟨5⟩5. Q.E.D.

Proof: From ⟨5⟩2, ⟨4⟩1, and ⟨5⟩4.

⟨4⟩13. Q.E.D.

Proof: From ⟨4⟩2, ⟨4⟩8, ⟨4⟩9, ⟨4⟩10, ⟨4⟩11, and ⟨4⟩12.

⟨3⟩6. Case: $e \notin G_p.\mathsf{E}$

⟨4⟩1. $A' = A +\!\!+\, e$

⟨4⟩2. $G_{MC}(A') \neq \bot$

⟨5⟩1. $G_{MC}(A) \neq \bot$

Proof: From loop invariant 1.

⟨5⟩2. $dom(G_{MC}(A).\mathtt{symb}; [r]) \subseteq G_{MC}(A).\mathsf{E}$

Proof: From $\mathsf{next}_P(G(S)) = e$, we have $dom(G(S).\mathtt{symb}; [e]) \subseteq G(S).\mathsf{E}$. The result follows from $G(S) \sqsubseteq G_{MC}(A)$.

⟨5⟩3. Case: $e \in \mathsf{R}$

Proof: From definition of $\rightsquigarrow$ and ⟨4⟩1.

⟨5⟩4. Case: $e \in \mathsf{W}$

⟨6⟩1. There is a matching open read e_o

⟨6⟩2. $e_o \in A$

⟨7⟩1. $e_o \notin G_p$

Proof: From ⟨3⟩6 and e_o being the matching read of e.

⟨7⟩2. $G(S) \sqsubseteq G_{MC}(A)$

Proof: From ⟨3⟩3.

⟨7⟩3. Q.E.D.

Proof: From ⟨7⟩1 and ⟨7⟩2.

⟨6⟩3. $e_0 = \textsc{last}(A)$

Proof: From ⟨6⟩1, ⟨6⟩2, and loop invariant 5.

⟨6⟩4. Q.E.D.

Proof: From ⟨6⟩3, ⟨4⟩1, and definition of $\rightsquigarrow$.

⟨5⟩5. Q.E.D.

Proof: From ⟨5⟩2, ⟨5⟩3, and ⟨5⟩4.

⟨4⟩3. $\mathsf{consistent}_{\mathrm{SYM}}(G_{MC}(A'))$

Proof: From ⟨2⟩3 and Corollary E.10.

⟨4⟩4. $\mathsf{next}_P(G) \in G_{MC}(A')$

Proof: From ⟨4⟩1.

⟨4⟩5. $G(S) \sqsubset G_{MC}(A')$

⟨5⟩1. $G(S) \sqsubseteq G_{MC}(A')$

Proof: From ⟨3⟩3 and ⟨4⟩1.

⟨5⟩2. $G(S) \neq G_{MC}(A')$

Proof: From ⟨4⟩4.

⟨5⟩3. Q.E.D.

⟨4⟩6. $\langle -|\textsc{Apply}(S).\mathsf{E}|, |G_{MC}(A').\mathsf{E}|\rangle < M$

⟨5⟩1. $G(S) \sqsupseteq G_{S_0}$

Proof: From ⟨3⟩3.

⟨5⟩2. Case: $G(S) \neq G_{S_0}$

Proof: From ⟨5⟩1 and ⟨5⟩2, it is $G(S) \sqsubset G_{S_0}$ and therefore $-|G(S).\mathsf{E}| < -|G_{S_0}|.\mathsf{E}$.

⟨5⟩3. Case: $G(S) = G_{S_0}$

⟨6⟩1. $S = S_0$

PROOF: From loop invariant 3.

⟨6⟩2. $A' = [e_0]$

PROOF: From loop invariant 2, $\mathsf{next}_P(G_{S_0}) = e_0$, and ⟨4⟩1.

⟨6⟩3. Q.E.D.

PROOF: It is $G_{MC}(A').\mathsf{E} = G_p.\mathsf{E} \cup \{e_0\} \subseteq G_t.\mathsf{E} \setminus \{w\}$ and therefore $|G_{MC}(A').\mathsf{E}| < |G_t.\mathsf{E}|$.

⟨5⟩4. Q.E.D.

PROOF: Cases ⟨5⟩2 and ⟨5⟩3 are exhaustive.

⟨4⟩7. 1. $S_0 \sqsubset S'$, $\textsc{Apply}(S') = \langle G(S'), <_{S'} \rangle$, and $G(S) \sqsubset G(S') \sqsubseteq G_{MC}(A')$

2. $<_S \mathbin{+\!\!+} \mathsf{next}_P(G(S)) \sqsubseteq <_{S'}$ and $\textsc{last}(S')$ either adds or revisits $\mathsf{next}_P(G(S))$

3. If $\bar{S} \sqsupset S_0$ and $\textsc{Apply}(\bar{S}) \sqsupseteq G_{MC}(A')$, then $\bar{S} \sqsupseteq S'$

PROOF: From ⟨4⟩5, ⟨4⟩3, ⟨4⟩4, ⟨4⟩6, and ⟨1⟩1.

⟨4⟩8. $S' \neq S_0$

PROOF: From ⟨4⟩7:1.

⟨4⟩9. $S_0 \sqsubset S' \land G_{S_0} \sqsubset G(S') \sqsubseteq G_{MC}(A') \land <_{S_0} \mathbin{+\!\!+} e_0 \sqsubseteq <'_S$

PROOF: From ⟨4⟩7 and ⟨3⟩3.

⟨4⟩10. $A' = \mathsf{sort}_{<_{S'}}(G(S').\mathsf{E} \setminus G_p.\mathsf{E})$ and $w \notin A'$

⟨5⟩1. $A = \mathsf{sort}_{<_S}(G(S).\mathsf{E} \setminus G_p.\mathsf{E})$ and $w \notin A$

PROOF: From loop invariant 4.

⟨5⟩2. $w \notin A'$

PROOF: From ⟨5⟩1 and $e \neq w$ (⟨3⟩2).

⟨5⟩3. $G(S').\mathsf{E} \setminus G(S).\mathsf{E} \subseteq G_p.\mathsf{E} \cup \{e\}$

From $A \subseteq G(S).\mathsf{E}$ (⟨5⟩1), ⟨4⟩9, and ⟨4⟩1.

⟨5⟩4. $G(S').\mathsf{E} \setminus G_p.\mathsf{E} = (G(S).\mathsf{E} \setminus G_p.\mathsf{E}) \cup \{e\}$

PROOF: From ⟨5⟩3.

⟨5⟩5. Q.E.D.

PROOF: From ⟨5⟩4, $<_S \sqsubseteq <_{S'}$ (⟨4⟩7:2), ⟨5⟩1, and ⟨4⟩1.

⟨4⟩11. If $\exists e_o \in A'.\, \mathit{open}(G(S'), e_o)$, then $\Rightarrow e_o = \textsc{last}(A')$

PROOF: From ⟨4⟩7 $\textsc{last}(S')$ either adds or revisits e and from Prop. E.22, if there is an open read in G', it is just added or revisited. Therefore, if there is an open read, it is $e = \textsc{last}(A')$.

⟨4⟩12. If $\hat{S} \sqsupset S_0$ and $\textsc{Apply}(\bar{S}) \sqsupseteq G_t$, then $\bar{A} \sqsupseteq A'$ and $\bar{S} \sqsupseteq S'$

⟨5⟩1. ASSUME: $\hat{S} \sqsupset S_0$ and $\textsc{Apply}(\bar{S}) \sqsupseteq G_t$

⟨5⟩2. $\bar{A} \sqsupseteq A$ and $\bar{S} \sqsupseteq S$

PROOF: From loop invariant 6 and ⟨2⟩6 (2).

⟨5⟩3. $G_{MC}(A) \sqsubseteq \bar{G}$

PROOF: $G_{MC}(A) \sqsubseteq G_{MC}(\bar{A}) = \bar{G}$ ($A \sqsubseteq \bar{A}$ from ⟨5⟩2).

⟨5⟩4. $\bar{S} \sqsupset S$

PROOF: From ⟨5⟩2 and $\mathsf{next}_P(G) = e \neq w = \mathsf{next}_P(\bar{G})$ (⟨3⟩2).

⟨5⟩5. $e \in \bar{G}.\mathsf{E}$ and $<_S \mathbin{+\!\!+} e \sqsubseteq \bar{<}$

PROOF: From ⟨5⟩3, ⟨5⟩4, and Prop. E.23.

⟨5⟩6. $\bar{A} \sqsupseteq A'$

PROOF: From ⟨5⟩2, ⟨5⟩5, ⟨3⟩6, and definition of $\bar{A}$ (⟨2⟩6:5).

⟨5⟩7. $G_{MC}(A') \sqsubseteq \bar{G}$

PROOF: From ⟨5⟩6.

⟨5⟩8. $\bar{S} \sqsupseteq S'$

PROOF: From ⟨4⟩7 (3), ⟨5⟩4, and ⟨5⟩8.

⟨5⟩9. Q.E.D.

PROOF: From ⟨5⟩6 and ⟨5⟩8.

⟨4⟩13. Q.E.D.

PROOF: From ⟨4⟩2, ⟨4⟩8, ⟨4⟩9, ⟨4⟩10, ⟨4⟩11, and ⟨4⟩12.

⟨2⟩10. LET: 1. $\langle S_l, A_l\rangle$ be the values at the last iteration.
2. $\langle G_l, <_l\rangle \triangleq \text{APPLY}(S_l)$
3. $P \triangleq dom((G_l \uplus \{w\}).\mathsf{porf}; [\{w\}])$
4. S' be the return value in Line 18

PROOF: The loop will eventually terminate because at each iteration, the size of the execution increases ($G(S) \sqsubset G(S')$), and the program semantics only contain executions of finite size.

⟨2⟩11. $\mathsf{Inv}(S_l, A_l)$

PROOF: From ⟨2⟩9.

⟨2⟩12. $\mathsf{next}_P(G_l) = w$

PROOF: The test in Line 17 succeeds.

⟨2⟩13. $S_0 \neq S_l$

PROOF: $\mathsf{next}_P(G_{S_0}) = e_0 \neq w = \mathsf{next}_P(G_l)$ (⟨2⟩13).

⟨2⟩14. $G_l = \text{MAXCOMPLETION}(G_p, A_l)$

PROOF: From ⟨2⟩13, ⟨2⟩11:3, ⟨2⟩12, and definition of w (⟨2⟩2): all event in G_p must have already been added.

⟨2⟩15. $\{e' \in G_l.\mathsf{E} \mid e' <_l e_0\} = G_{S_0}.\mathsf{E}$

PROOF: From $G_{S_0} \sqsubseteq G_l$ (⟨2⟩14), ⟨2⟩13, and Prop. E.23.

⟨2⟩16. $A_l = \mathsf{sort}_<(\{e' \in G_l.\mathsf{E} \setminus P \mid e_0 \leq e'\})<_l$

PROOF: From ⟨2⟩15 and ⟨2⟩11:4.

⟨2⟩17. LET: $G' : G_p \overset{w@p}{\rightarrow} \overset{e_0@w}{\rightarrow} G'$

⟨2⟩18. $\mathsf{consistent}_{\mathsf{SYM}}(G')$

PROOF: From Corollary E.11 ($G' \sqsubseteq G_t$).

⟨2⟩19. LET: S' be the production sequence in Line 18

⟨2⟩20. $\text{APPLY}(S') = \langle G', <'\rangle$

PROOF: The revisit step is enabled: ⟨2⟩12, ⟨2⟩14, ⟨2⟩16, and ⟨2⟩18.

⟨2⟩21. $S_0 \sqsubset S'$

PROOF: $S_0 \sqsubseteq S_l$ (⟨2⟩13 and ⟨2⟩11:3) and $S_l \sqsubset S'$.

⟨2⟩22. $\text{LAST}(S')$ revisits e_0

⟨2⟩23. $<_{S_0} +\!\!+\; e_0 \sqsubseteq <'$

PROOF: From ⟨2⟩11:3 and ⟨2⟩21.

⟨2⟩24. If $\hat{S} \sqsupset S_0$ and $\text{APPLY}(\hat{S}) \sqsupseteq G_t$, then $\hat{S} \sqsupseteq S'$

⟨3⟩1. $\bar{S} = S_l$

PROOF: It is $S_l \sqsubseteq \bar{S}$ and $G_l \sqsubseteq \bar{G}$ (⟨2⟩11:6), and if $S_l \sqsubset \bar{S}$, we have $w \in \bar{G}$ from Prop. E.23, which contradicts $\mathsf{next}_P(\bar{G}) = w$.

⟨3⟩2. Q.E.D.

PROOF: From ⟨3⟩1 and $S_l \sqsubseteq S'$.

⟨2⟩25. Q.E.D.

PROOF: From ⟨2⟩20, ⟨2⟩21, ⟨2⟩22, ⟨2⟩23, and ⟨2⟩24.

⟨1⟩4. Q.E.D.

PROOF: From ⟨1⟩1 and the fact that cases ⟨1⟩2 and ⟨1⟩3 are exhaustive. □

THEOREM E.27. *Given an execution $G_f \in [\![P]\!]_M^{\mathsf{Annot}}$ such that* ACYCLIC$(G_f.\mathsf{cb}_{SYM})$, *PRODUCTIONSEQUENCE(G_f) will return the unique production sequence S such that APPLY$(S) = G_f$.*

PROOF. We will show that $\textsc{ProductionSequence}(G_f)$ returns a production sequence S_f such that $\textsc{Apply}(S_f) = G_f$, and if there is a production sequence $\hat{S}$ such that $\textsc{Apply}(\hat{S}) = G_f$, it is $\hat{S} = S_f$. We will prove that at the beginning of every iteration of $\textsc{ProductionSequence}(G_f)$ where $G \triangleq \textsc{Apply}(S)$, it is $S \sqsubseteq \hat{S}$ and either $G = G_f$ or $G \sqsubset G_f$ and $G \sqsubset G'$ where $G' = \textsc{Apply}(S')$ and S' is the value of S at the end of the iteration.

Since G_f is full, $dom(G_f.\mathsf{symb}; [G_f.\mathsf{E}]) \subseteq G_f.\mathsf{E}$. From the hypothesis, it is also $\textsc{acyclic}(G_f.\mathsf{cb}_{\mathsf{SYM}})$, and therefore, from Prop. E.12, $G_\emptyset \sqsubseteq G_f$. Therefore, from the first iteration it is $\emptyset \sqsubseteq \hat{S}$ and either $G_f = G_\emptyset$ or $G_\emptyset \sqsubset G_f$. For any other iteration before the loop terminates with $G \sqsubset G_f$ and $S \sqsubseteq \hat{S}$, we have $\mathsf{next}_P(G) \in G_f.\mathsf{E}$ because G_f is full, and $\mathsf{consistent}_{\mathsf{SYM}}(G_f)$ from the hypothesis, and therefore $G \sqsubset G' \sqsubseteq G_t$ and $S' \sqsubseteq \hat{S}$ from Lemma E.26.

The loop will terminate since the program semantics only contain executions of finite size and $\mathsf{consistent}_{\mathsf{SYM}}(G)$. For the final value S_f, it is $\textsc{Apply}(S_f) = G_f$, and $S_f \sqsubseteq \hat{S}$. Since G_f is full, there is no other step to be taken from S_f, and therefore $\hat{S} = S_f$. □

F External Symmetries

Lemma F.1. *Let G be an execution s.t.* $\mathsf{consistent}_{SYM}(G)$ *with a* cb_{SYM} *cycle. Then G has a* $\mathsf{po} \cup \mathsf{rf} \cup \mathsf{co}$ *cycle.*

Proof. Assume the opposite. Let C be the set of $G.\mathsf{po} \cup G.\mathsf{rf} \cup G.\mathsf{co} \cup G.\mathsf{symb}$ cycles. Since there is a $G.\mathsf{cb}_{\mathsf{SYM}}$ cycle, C is non-empty. Let c be a cycle of C with the minimal number of $G.\mathsf{symb}$ edges. Since $G.\mathsf{po} \cup G.\mathsf{rf} \cup G.\mathsf{co}$ is acyclic, c has at least one $G.\mathsf{symb}$ edge $e = \langle x, y \rangle$. It must be that e is between read events, otherwise it can be rewritten as $G.\mathsf{co}$ ($G.\mathsf{symb} \cap G.\mathsf{co}^{-1} = \emptyset$), contradicting that c has the minimal number $G.\mathsf{symb}$ edges among the cycles of C. Edge e cannot be the only edge of c since $G.\mathsf{symb}$ is irreflexive. Let p be the path of c excluding the edge e. If p ends with a $G.\mathsf{po}$ edge, then there is also a cycle that uses p to enter the thread of y instead of x, i.e., it has one less $G.\mathsf{symb}$ edge, and thus again contradicts the hypothesis about c. For the same reason p cannot end with a $G.\mathsf{symb}$ edge: $G.\mathsf{symb}$ is transitive and the cycle could be written by combining two consecutive $G.\mathsf{symb}$ edges. Therefore, p ends with a $G.\mathsf{rf}$ edge. It must be that x and y read from different writes, otherwise c can be rewritten by using the $G.\mathsf{rf}$ edge that ends in y, avoiding again edge e. Since x and y read from different writes, $\langle x, y \rangle \in G.\mathsf{symb}$, and $G.\mathsf{symb} \cap G.\mathsf{eco}^{-1} = \emptyset$, it is $\langle x, y \rangle \in G.\mathsf{eco}$, and thus $\langle x, y \rangle \in G.\mathsf{rb}; G.\mathsf{rf}$. Therefore the cycle c is $p'; G.\mathsf{rf}; [x]; G.\mathsf{rf}^{-1}; G.\mathsf{co}; G.\mathsf{rf}; [y]$ which can be rewritten as $p'; G.\mathsf{co}; G.\mathsf{rf}$. This again contradicts the assumption that c is the cycle of C with the minimal number of $G.\mathsf{symb}$ edges, concluding the proof. □

Proposition F.2 ($\mathsf{cb}_{\mathsf{SYM}}$ cycle). *If there is an execution $G \in [\![P]\!]^{\mathsf{Annot}}_{SYM}$ with a $G.\mathsf{cb}_{SYM}$ cycle, then there is an execution $G' \in [\![P]\!]^{\mathsf{Annot}}_{SYM}$ such that* IRREFLEXIVE$(G'.\mathsf{cb}_{SYM})$ *and G' has a* $\mathsf{po} \cup \mathsf{rf} \cup \mathsf{co}$ *cycle.*

Proof. From Lemma F.1, G has a $\mathsf{po} \cup \mathsf{rf} \cup \mathsf{co}$ cycle. Let G' be a maximal porf-prefix of G s.t. G' has no $\mathsf{po} \cup \mathsf{rf} \cup \mathsf{co}$ cycle, i.e., extending by any other event of G introduces such a cycle. Let e be any such event and G'' the resulting execution. Observe that e must be a write: adding any read event cannot introduce such a cycle. From Lemma F.1, G' has no $\mathsf{cb}_{\mathsf{SYM}}$ cycle. Since G' and G'' only differ by the write e, G'' also has no $\mathsf{cb}_{\mathsf{SYM}}$ cycle: any such cycle must include a po edge from an event x to e followed by a symb edge to y, which can also be rewritten to avoid e (there exists an $\mathsf{symb}; \mathsf{po}$ edge from x to y). We can now sort any symmetric threads that are incomparable w.r.t. to eco (lexicographically, in po order, i.e., one is a prefix of the other) in decreasing thread length. This reordering cannot introduce any $\mathsf{cb}_{\mathsf{SYM}}$-cycles (no symb edge is added), and, by construction, for the resulting execution, symb is backward-closed and respects eco. Therefore, we can maximally extend it in left-to-right thread order, and obtain a consistent, $\mathsf{cb}_{\mathsf{SYM}}$-acyclic execution whose symb respects eco and has a $\mathsf{po} \cup \mathsf{rf} \cup \mathsf{co}$ cycle. □

G Experiments

In what follows, "Execs" denotes the number of complete executions explored, and "Blocked" denotes the number of blocked executions explored (e.g., due to an `assume` failing). We do not report times as all tools are based on the same implementation, and time is directly proportionate to the execution number. For the comparison with Nidhugg, we only use non-data-structure benchmarks because Nidhugg does not support hazard pointers, as a result of which it cannot handle our queue benchmarks.

Table 1. Multiset client

	SR		TruSt		DPOR+IS		DPOR+SR		Spore	
	Execs	*Blocked*	*Execs*	*Blocked*	*Execs*	*Blocked*	*Execs*	*Blocked*	*Execs*	*Blocked*
msqueue(1)	1	0	1	0	1	0	1	0	1	0
msqueue(2)	37	0	3	1	2	1	3	1	2	0
msqueue(3)	🕒	🕒	26	38	6	14	13	19	3	0
msqueue(4)	🕒	🕒	158	694	22	102	41	179	6	4
msqueue(5)	🕒	🕒	4638	30 402	114	1230	397	2623	10	9
msqueue(6)	🕒	🕒	43 434	783 193	618	16 740	1284	23 368	20	69
msqueue(7)	🕒	🕒	🕒	🕒	4560	278 432	22 469	564 441	35	155
msqueue(8)	🕒	🕒	🕒	🕒	32 760	5 490 520	🕒	🕒	70	965
dglmqueue(1)	1	0	1	0	1	0	1	0	1	0
dglmqueue(2)	10	0	4	0	3	0	4	0	3	0
dglmqueue(3)	32 313	0	34	32	8	12	17	16	4	0
dglmqueue(4)	🕒	🕒	286	416	50	92	73	106	13	3
dglmqueue(5)	🕒	🕒	10 926	20 964	246	1188	921	1790	21	4
dglmqueue(6)	🕒	🕒	145 926	411 695	2454	14 316	4206	12 151	73	36
dglmqueue(7)	🕒	🕒	🕒	🕒	18 744	229 672	118 857	336 550	136	55
dglmqueue(8)	🕒	🕒	🕒	🕒	263 064	3 829 296	573 105	2 508 330	501	390
folqueue(1)	1	0	1	0	1	0	1	0	1	0
folqueue(2)	357	0	4	2	3	1	4	2	3	1
folqueue(3)	🕒	🕒	48	24	12	6	48	24	12	6
folqueue(4)	🕒	🕒	358	1418	58	204	182	718	30	81
folqueue(5)	🕒	🕒	15 318	72 069	414	2002	7722	36 249	210	822
folqueue(6)	🕒	🕒	170 514	3 474 177	2796	65 521	29 406	611 749	504	8488
folqueue(7)	🕒	🕒	🕒	🕒	28 968	931 053	🕒	🕒	5040	127 172
folqueue(8)	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒	11 880	1 367 168
treiber(1)	1	0	1	0	1	0	1	0	1	0
treiber(2)	2	0	2	0	2	0	2	0	2	0
treiber(3)	183	0	8	6	8	6	4	3	4	3
treiber(4)	13 190	0	38	32	38	32	10	9	10	9
treiber(5)	🕒	🕒	282	645	282	645	24	59	24	59
treiber(6)	🕒	🕒	2292	5736	2292	5736	68	191	68	191
treiber(7)	🕒	🕒	24 576	131 752	24 576	131 752	176	1066	176	1066
treiber(8)	🕒	🕒	292 920	1 683 556	292 920	1 683 556	546	3682	546	3682

Table 2. LIFO/FIFO client

	SR		TruSt		DPOR+IS		DPOR+SR		Spore	
	Execs	*Blocked*	*Execs*	*Blocked*	*Execs*	*Blocked*	*Execs*	*Blocked*	*Execs*	*Blocked*
msqueue(1)	🕒	🕒	181	316	15	54	181	316	15	16
msqueue(2)	🕒	🕒	1714	12 425	72	790	1714	12 425	72	313
msqueue(3)	🕒	🕒	121 832	1 093 531	672	12 772	60 916	548 202	336	2030
msqueue(4)	🕒	🕒	🕒	🕒	4344	272 652	🕒	🕒	930	14 463
msqueue(5)	🕒	🕒	🕒	🕒	49 560	5 787 820	🕒	🕒	3580	70 700
msqueue(6)	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒	8240	430 480
msqueue(7)	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒	28 740	1 835 111
msqueue(8)	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒
dglmqueue(1)	🕒	🕒	315	399	27	63	315	399	27	28
dglmqueue(2)	🕒	🕒	4275	9592	252	892	4275	9592	252	435
dglmqueue(3)	🕒	🕒	414 904	1 057 713	2040	14 944	207 452	529 850	1020	2674
dglmqueue(4)	🕒	🕒	🕒	🕒	27 108	270 608	1 993 212	8 081 770	6606	19 707
dglmqueue(5)	🕒	🕒	🕒	🕒	304 680	5 507 860	🕒	🕒	24 760	85 831
dglmqueue(6)	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒	134 060	519 354
dglmqueue(7)	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒	498 525	2 008 304
dglmqueue(8)	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒
folqueue(1)	🕒	🕒	642	251	69	32	642	251	69	32
folqueue(2)	🕒	🕒	6993	42 493	390	2619	6993	42 493	390	2124
folqueue(3)	🕒	🕒	754 830	4 717 584	5004	35 037	754 830	4 717 584	5004	29 122
folqueue(4)	🕒	🕒	🕒	🕒	38 352	1 797 068	🕒	🕒	18 288	536 895
folqueue(5)	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒
folqueue(6)	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒
folqueue(7)	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒
folqueue(8)	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒
treiber(1)	40 934	0	7	8	7	8	7	8	7	8
treiber(2)	🕒	🕒	56	95	56	95	56	95	56	95
treiber(3)	🕒	🕒	642	1793	642	1793	321	914	321	914
treiber(4)	🕒	🕒	7172	27 939	7172	27 939	1808	7255	1808	7255
treiber(5)	🕒	🕒	109 296	666 885	109 296	666 885	9148	58 610	9148	58 610
treiber(6)	🕒	🕒	🕒	🕒	🕒	🕒	45 590	387 936	45 590	387 936
treiber(7)	🕒	🕒	🕒	🕒	🕒	🕒	211 235	2 886 319	211 235	2 886 319
treiber(8)	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒

Table 3. Emptiness client

	SR		TruSt		DPOR+IS		DPOR+SR		Spore	
	Execs	*Blocked*	*Execs*	*Blocked*	*Execs*	*Blocked*	*Execs*	*Blocked*	*Execs*	*Blocked*
msqueue(1)	1	0	1	0	1	0	1	0	1	0
msqueue(2)	13 554	0	16	48	6	16	8	24	3	6
msqueue(3)	🕒	🕒	1368	15 350	90	1003	228	2574	15	90
msqueue(4)	🕒	🕒	🕒	🕒	2520	108 696	16 188	634 944	105	1967
msqueue(5)	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒	945	57 559
msqueue(6)	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒	10 395	2 195 070
msqueue(7)	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒
msqueue(8)	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒
dglmqueue(1)	1	0	1	0	1	0	1	0	1	0
dglmqueue(2)	17 146	0	16	40	6	13	8	20	3	4
dglmqueue(3)	🕒	🕒	1488	9990	102	611	248	1661	17	52
dglmqueue(4)	🕒	🕒	🕒	🕒	3672	47 640	20 680	311 868	153	904
dglmqueue(5)	🕒	🕒	🕒	🕒	238 680	6 154 014	🕒	🕒	1989	20 587
dglmqueue(6)	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒	35 115	601 822
dglmqueue(7)	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒
dglmqueue(8)	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒
folqueue(1)	1	0	1	0	1	0	1	0	1	0
folqueue(2)	🕒	🕒	20	54	8	18	20	54	8	18
folqueue(3)	🕒	🕒	2160	24 849	168	1786	2160	24 849	168	1465
folqueue(4)	🕒	🕒	🕒	🕒	6720	340 861	🕒	🕒	6720	236 771
folqueue(5)	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒
folqueue(6)	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒
folqueue(7)	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒
folqueue(8)	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒
treiber(1)	1	0	1	0	1	0	1	0	1	0
treiber(2)	927	0	10	6	10	6	5	3	5	3
treiber(3)	🕒	🕒	270	387	270	387	45	65	45	65
treiber(4)	🕒	🕒	13 992	38 536	13 992	38 536	583	1615	583	1615
treiber(5)	🕒	🕒	1 188 600	5 740 545	1 188 600	5 740 545	9905	48 052	9905	48 052
treiber(6)	🕒	🕒	🕒	🕒	🕒	🕒	209 141	1 694 732	209 141	1 694 732
treiber(7)	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒
treiber(8)	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒

Table 4. Non-data-structure benchmarks

	SR		TruSt		DPOR+IS		DPOR+SR		Spore	
	Execs	*Blocked*	*Execs*	*Blocked*	*Execs*	*Blocked*	*Execs*	*Blocked*	*Execs*	*Blocked*
ttaslock(1)	1	0	1	0	1	0	1	0	1	0
ttaslock(2)	🕒	🕒	4	0	4	0	2	0	2	0
ttaslock(3)	🕒	🕒	36	0	36	0	6	0	6	0
ttaslock(4)	🕒	🕒	576	0	576	0	24	0	24	0
ttaslock(5)	🕒	🕒	14 400	0	14 400	0	120	0	120	0
ttaslock(6)	🕒	🕒	518 400	0	518 400	0	720	0	720	0
ttaslock(7)	🕒	🕒	🕒	🕒	🕒	🕒	5040	0	5040	0
ttaslock(8)	🕒	🕒	🕒	🕒	🕒	🕒	40 320	0	40 320	0
twalock(1)	1	0	1	0	1	0	1	0	1	0
twalock(2)	🕒	🕒	4	0	4	0	2	0	2	0
twalock(3)	🕒	🕒	96	0	96	0	16	0	16	0
twalock(4)	🕒	🕒	6144	0	6144	0	256	0	256	0
twalock(5)	🕒	🕒	798 720	0	798 720	0	6656	0	6656	0
twalock(6)	🕒	🕒	🕒	🕒	🕒	🕒	252 928	0	252 928	0
twalock(7)	🕒	🕒	🕒	🕒	🕒	🕒	13 152 256	0	13 152 256	0
twalock(8)	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒
rdcss(1)	98 681	0	51	56	7	6	26	27	4	3
rdcss(2)	🕒	🕒	2296	4048	48	60	580	958	14	16
rdcss(3)	🕒	🕒	250 219	632 803	503	822	20 988	49 953	51	83
rdcss(4)	🕒	🕒	🕒	🕒	7302	14 686	1 139 613	3 677 091	194	428
rdcss(5)	🕒	🕒	🕒	🕒	138 787	330 889	🕒	🕒	772	2234
rdcss(6)	🕒	🕒	🕒	🕒	3 315 560	9 122 025	🕒	🕒	3212	11 874
rdcss(7)	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒	13 952	64 459
rdcss(8)	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒	63 150	357 813

Table 5. Nidhugg, TruSt and Spore on non-data-structure benchmarks

	Nidhugg			TruSt			Spore		
	Execs	*Blocked*	*Time*	*Execs*	*Blocked*	*Time*	*Execs*	*Blocked*	*Time*
`ttaslock(3,1)`	36	39	0.14	36	0	0.12	6	0	0.09
`ttaslock(4,1)`	576	1084	0.44	576	0	0.14	24	0	0.09
`ttaslock(5,1)`	14 400	42 845	12.97	14 400	0	1.41	120	0	0.12
`ttaslock(6,1)`	518 400	2 320 386	759.18	518 400	0	56.14	720	0	0.17
`ttaslock(7,1)`	🕒	🕒	🕒	🕒	🕒	🕒	5040	0	0.63
`ttaslock(8,1)`	🕒	🕒	🕒	🕒	🕒	🕒	40 320	0	5.21
`ttaslock(3,2)`	7134	11 223	4.12	7134	0	0.73	1189	0	0.21
`ttaslock(4,2)`	🕒	🕒	🕒	5 189 880	0	568.32	216 245	0	24.09
`ttaslock(5,2)`	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒
`ttaslock(6,2)`	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒
`twalock(3,1)`	96	0	0.14	96	0	0.35	16	0	0.34
`twalock(4,1)`	6144	0	1.71	6144	0	0.92	256	0	0.35
`twalock(5,1)`	798 720	0	263.18	798 720	0	95.88	6656	0	0.77
`twalock(6,1)`	🕒	🕒	🕒	🕒	🕒	🕒	252 928	0	16.68
`twalock(7,1)`	🕒	🕒	🕒	🕒	🕒	🕒	13 152 256	0	936.36
`twalock(3,2)`	84 936	0	25.58	84 936	0	10.12	14 156	0	1.75
`twalock(4,2)`	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒	🕒
`rdcss(1,2)`	51	59	0.19	51	56	0.12	4	3	0.11
`rdcss(2,2)`	2296	4754	2.80	2296	4048	0.62	14	16	0.11
`rdcss(3,2)`	250 219	821 903	536.00	250 219	632 803	62.57	51	83	0.13
`rdcss(4,2)`	🕒	🕒	🕒	🕒	🕒	🕒	194	428	0.21
`rdcss(5,2)`	🕒	🕒	🕒	🕒	🕒	🕒	772	2234	0.64
`rdcss(6,2)`	🕒	🕒	🕒	🕒	🕒	🕒	3212	11 874	3.00
`rdcss(7,2)`	🕒	🕒	🕒	🕒	🕒	🕒	13 952	64 459	16.58
`rdcss(8,2)`	🕒	🕒	🕒	🕒	🕒	🕒	63 150	357 813	94.05

H Client code

H.1 Multiset client

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <stdbool.h>
#include <stdatomic.h>
#include <assert.h>

#include "../../../../lib/queue-wrapper.h"

#ifndef MAX_THREADS
# define MAX_THREADS 32
#endif

#ifdef READERS
#define DEFAULT_READERS (READERS)
#else
#define DEFAULT_READERS 1
#endif

#ifdef WRITERS
#define DEFAULT_WRITERS (WRITERS)
#else
#define DEFAULT_WRITERS 1
#endif

#ifndef HP_THREAD_LIMIT
# define HP_THREAD_LIMIT 32
#endif

int readers = DEFAULT_READERS, writers = DEFAULT_WRITERS;

queue_t *queue;
queue_t myqueue;
int num_threads;

unsigned int input[MAX_THREADS + 1];
unsigned int output[MAX_THREADS + 1];

int __thread tid;

__VERIFIER_hp_t hps[MAX_THREADS + 1][HP_THREAD_LIMIT];
int __thread __hp_index;

/* Keep track of how many readers failed */
bool failed[DEFAULT_READERS];

void set_thread_num(int i)
{
  tid = i;
}

int get_thread_num()
{
  return tid;
}

__VERIFIER_hp_t *get_free_hp()
{
  int index = __hp_index++;
  assert(index < HP_THREAD_LIMIT);
  return &hps[tid][index];
}

void *threadW(void *param)
{
  int pid = (intptr_t ) param;
  set_thread_num(pid);

  input[pid] = pid + 42;
  enqueue(queue, input[pid]);
  return NULL;
}

void *threadR(void *param)
{
  int pid = (intptr_t ) param;

  set_thread_num(pid);

  /* UB if we mod with READERS == 0, but that's OK because
   * then this function will not be executed */
  failed[pid % DEFAULT_READERS] = !dequeue(queue, &output[pid]);
  return NULL;
}

int main()
{
  pthread_t threads[MAX_THREADS + 1];
  unsigned int in_sum = 0, out_sum = 0;
  int i = 0;

  queue = &myqueue;
  num_threads = readers + writers;

  init_queue(queue, num_threads);

  ++i;
  for (int j = 0; j < writers; j++, i++) {
    if (j == 0)
      pthread_create(&threads[i], NULL, threadW, (void *) (intptr_t) i);
    else
      threads[i] = __VERIFIER_spawn_symmetric(threadW, (void *)(intptr_t) i, threads[i-1]);
  }
  for (int j = 0; j < readers; j++, i++) {
    if (j == 0)
      pthread_create(&threads[i], NULL, threadR, (void *) (intptr_t) i);
    else
      threads[i] = __VERIFIER_spawn_symmetric(threadR, (void *)(intptr_t) i, threads[i-1]);
  }

  i = 1;
  for (int j = 0; j < writers; j++, i++) {
    if (j == 0)
      pthread_join(threads[i], NULL);
    else
      __VERIFIER_join_symmetric(threads[i]);
  }
  for (int j = 0; j < readers; j++, i++) {
    if (j == 0)
      pthread_join(threads[i], NULL);
    else
      __VERIFIER_join_symmetric(threads[i]);
  }

#ifdef PRINT_INFO
  printf("---\n");
  for (i = 1; i <= num_threads; i++)
    printf("input[%d] = %u, output[%d] = %u\n", i, input[i], i, output[i]);
#endif

  /* Dequeue whatever is left in the queue */
  unsigned tmp;
  while (dequeue(queue, &tmp))
    out_sum += tmp;

  /* Ensure that in_sum == out_sum */
  for (i = 1; i <= num_threads; i++) {
    in_sum += input[i];
    out_sum += output[i];
  }
  assert(in_sum == out_sum);

  return 0;
}
```

H.2 FIFO client

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <stdbool.h>
#include <stdatomic.h>
#include <assert.h>

#include "../../../../lib/queue-wrapper.h"

#ifndef MAX_THREADS
# define MAX_THREADS 32
#endif

#ifndef HP_THREAD_LIMIT
# define HP_THREAD_LIMIT 32
#endif

#ifdef NOISE_ENQ
#define DEFAULT_NOISE_ENQ (NOISE_ENQ)
#else
#define DEFAULT_NOISE_ENQ 1
#endif

#ifdef NOISE_DEQ
#define DEFAULT_NOISE_DEQ (NOISE_DEQ)
#else
#define DEFAULT_NOISE_DEQ 0
#endif

queue_t *queue;
queue_t myqueue;
int num_threads;

int __thread tid;

__VERIFIER_hp_t hps[MAX_THREADS + 1][HP_THREAD_LIMIT];
int __thread __hp_index;

void set_thread_num(int i)
{
  tid = i;
}

int get_thread_num()
{
  return tid;
}

__VERIFIER_hp_t *get_free_hp()
{
  int index = __hp_index++;
  assert(index < HP_THREAD_LIMIT);
  return &hps[tid][index];
}

void *thread_enq(void *param)
{
  int pid = (intptr_t ) param;
  set_thread_num(pid);
  enqueue(queue, 1);
  enqueue(queue, 2);
  return NULL;
}

void *thread_deq(void *param)
{
  int pid = (intptr_t ) param;
  unsigned dequeued[2];

  set_thread_num(pid);

  /* Ensure FIFO */
  __VERIFIER_assume(dequeue(queue, &dequeued[0]));
  __VERIFIER_assume(dequeue(queue, &dequeued[1]));
  assert(!(dequeued[0] == 2 && dequeued[1] == 1));
  return NULL;
}

void *noise_enq(void *param)
{
  int pid = (intptr_t ) param;

  set_thread_num(pid);
  enqueue(queue, 0);
  return NULL;
}

void *noise_deq(void *param)
{
  int pid = (intptr_t ) param;
  unsigned val;

  set_thread_num(pid);
  dequeue(queue, &val);
  return NULL;
}

int main()
{
  pthread_t te, td, noise[DEFAULT_NOISE_ENQ + DEFAULT_NOISE_DEQ + 2];

  queue = &myqueue;
  num_threads = 2 + DEFAULT_NOISE_ENQ + DEFAULT_NOISE_DEQ;

  init_queue(queue, num_threads);

  pthread_create(&te, NULL, thread_enq, (void *) (intptr_t) 1);
  pthread_create(&td, NULL, thread_deq, (void *) (intptr_t) 2);
  int i = 1;
  for (int j = 1; j <= DEFAULT_NOISE_ENQ; j++, i++) {
              if (j == 1)
                      pthread_create(&noise[i], NULL, noise_enq, (void *) (intptr_t) 2 + i);
              else
                      noise[i] = __VERIFIER_spawn_symmetric(noise_enq, (void *) (intptr_t) 2
      + i, noise[i-1]);
        }
  for (int j = 1; j <= DEFAULT_NOISE_DEQ; j++, i++) {
              if (j == 1)
                      pthread_create(&noise[i], NULL, noise_deq, (void *) (intptr_t) 2 + i);
              else
                      noise[i] = __VERIFIER_spawn_symmetric(noise_deq, (void *) (intptr_t) 2
      + i, noise[i-1]);
  }

  pthread_join(te, NULL);
  pthread_join(td, NULL);
  i = 1;
  for (int j = 1; j <= DEFAULT_NOISE_ENQ; j++, i++) {
              if (j == 1)
```

```
                        pthread_join(noise[i], NULL);
                else
                        __VERIFIER_join_symmetric(noise[i]);
  }
  for (int j = 1; j <= DEFAULT_NOISE_DEQ; j++, i++) {
                if (j == 1)
                        pthread_join(noise[i], NULL);
                else
                        __VERIFIER_join_symmetric(noise[i]);
  }
  return 0;
}
```

H.3 Emptiness client

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <stdbool.h>
#include <stdatomic.h>
#include <assert.h>

#include "../../../../lib/queue-wrapper.h"

#ifndef MAX_THREADS
# define MAX_THREADS 32
#endif

#ifndef HP_THREAD_LIMIT
# define HP_THREAD_LIMIT 32
#endif

#ifndef N
# define N 2
#endif

queue_t *queue;
queue_t myqueue;
int num_threads;

int __thread tid;

__VERIFIER_hp_t hps[MAX_THREADS + 1][HP_THREAD_LIMIT];
int __thread __hp_index;

void set_thread_num(int i)
{
  tid = i;
}

int get_thread_num()
{
  return tid;
}

__VERIFIER_hp_t *get_free_hp()
{
  int index = __hp_index++;
  assert(index < HP_THREAD_LIMIT);
  return &hps[tid][index];
}

void *thread_n(void *param)
{
  int pid = (intptr_t ) param;

  set_thread_num(pid);

  unsigned tmp;
  enqueue(queue, 1);
  assert(dequeue(queue, &tmp));
  return NULL;
}

int main()
{
  pthread_t threads[N + 1];

  queue = &myqueue;
  num_threads = N;

  init_queue(queue, num_threads);

  for (int i = 0; i < num_threads; i++) {
		if (i == 0)
			pthread_create(&threads[i], NULL, thread_n, (void *) (intptr_t) i);
		else
			threads[i] = __VERIFIER_spawn_symmetric(thread_n, (void *) (intptr_t)
     i, threads[i-1]);
  }

  for (int i = 0; i < num_threads; i++) {
		if (i == 0)
			pthread_join(threads[i], NULL);
		else
			__VERIFIER_join_symmetric(threads[i]);
  }

  return 0;
}
```

H.4 Mutex client

```
#include <stdlib.h>
#include <pthread.h>
#include <stdatomic.h>
#include <genmc.h>
#include <assert.h>

#include "../../../../lib/lock-wrapper.h"

#ifndef N
# define N 2
#endif

#ifndef M
# define M 1
#endif

int shared;
lock_t lock;

void *thread_n(void *arg)
{
  intptr_t index = ((intptr_t) arg);

  for (int i = 0u; i < M; i++) {
    lock_acquire(&lock);

    shared = index;
    int r = shared;
    assert(r == index);

    lock_release(&lock);
  }
  return NULL;
}
```

```
int main()
{
  pthread_t t[N];

  lock_init(&lock);
  for (int i = 0u; i < N; i++) {
    if (i == 0)
      pthread_create(&t[i], NULL, thread_n, (void *) (intptr_t) i);
    else
      t[i] = __VERIFIER_spawn_symmetric(thread_n, (void *)(intptr_t) i, t[i-1]);
  }
  for (int i = 0u; i < N; i++) {
    if (i == 0)
      pthread_join(t[i], NULL);
    else
      __VERIFIER_join_symmetric(t[i]);
  }

  return 0;
}
```

H.5 RDCSS client

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <stdatomic.h>
#include <genmc.h>

#include "rdcss.h"

#ifdef READERS
#define DEFAULT_READERS (READERS)
#else
#define DEFAULT_READERS 0
#endif

#ifdef WRITERS
#define DEFAULT_WRITERS (WRITERS)
#else
#define DEFAULT_WRITERS 0
#endif

#ifdef RDRW
#define DEFAULT_RDWR (RDRW)
#else
#define DEFAULT_RDWR 0
#endif

int readers = DEFAULT_READERS, writers = DEFAULT_WRITERS, rdwr = DEFAULT_RDWR;

_Atomic(value_t) x;
_Atomic(value_t) y;

void *threadW(void *param)
{
  descriptor_t *desc = malloc(sizeof(descriptor_t));
  desc->o1 = MAKE_INT_VAL(0);
  desc->o2 = MAKE_INT_VAL(0);
  desc->n2 = MAKE_INT_VAL(42);
  desc->a1 = &x;
  desc->a2 = &y;

  rdcss(desc);
  return NULL;
}

void *threadR(void *param)
{
  rdcss_read(&y);
  return NULL;
}

void *threadRW(void *param)
{
  value_t v = rdcss_read(&y);

  descriptor_t *desc = malloc(sizeof(descriptor_t));
  desc->o1 = MAKE_INT_VAL(0);
  desc->o2 = v;
  desc->n2 = MAKE_INT_VAL(GET_INT_VAL(v) + 1);
  desc->a1 = &x;
  desc->a2 = &y;
  rdcss(desc);
  return NULL;
}

int main()
{
  pthread_t tr[DEFAULT_READERS], tw[DEFAULT_WRITERS], trw[DEFAULT_RDWR];

  for (int i = 0; i < writers; i++) {
    if (i == 0)
      pthread_create(&tw[i], NULL, threadW, NULL);
    else
      tw[i] = __VERIFIER_spawn_symmetric(threadW, NULL, tw[i-1]);
  }
  for (int i = 0; i < readers; i++) {
    if (i == 0)
      pthread_create(&tr[i], NULL, threadR, NULL);
    else
      tr[i] = __VERIFIER_spawn_symmetric(threadR, NULL, tr[i-1]);
  }
  for (int i = 0; i < rdwr; i++) {
    if (i == 0)
      pthread_create(&trw[i], NULL, threadRW, NULL);
    else
      trw[i] = __VERIFIER_spawn_symmetric(threadRW, NULL, trw[i-1]);
  }

  return 0;
}
```

Received 2023-11-16; accepted 2024-03-31