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:

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

\[
\text{fetch}_\text{add}(x, 1) \parallel \ldots \parallel \text{fetch}_\text{add}(x, 1) \quad \text{(FAIS)}
\]

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:

\[
a := 1 \parallel b := 2 \parallel \ldots \parallel z := 26 \quad \text{(ARRAY)}
\]

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:

\[
i := \text{fetch}_\text{add}(x, 1) \parallel a[i] := i \quad \ldots \quad i := \text{fetch}_\text{add}(x, 1) \parallel a[i] := i \quad \text{(FAIS+ARRAY)}
\]

While DPOR explores $N!$ executions for FAIS+ARRAY (due to the conflicting fetch_adds), 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.

\section{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.

\section{3} We present Spore in detail and prove its correctness.

\section{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.

$T_1$: $x := 1$  \(\parallel\) $T_2$: $r_2 := x$  \(\parallel\) $T_3$: $r_3 := x$  \((w+r+r)\)

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 \(\text{init}\) through the intermediate graphs \(A\), \(B\), and \(C\).

![Diagram of DPOR exploration](image-url)

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 $T_2$ 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 $r_f$ as a relation from the write to the read event; (2) assume a special initialization event $\text{init}$, which initializes every location with 0 and is thus $\text{po}$-before all other events and $\text{co}$-before all other write events; (3) we do not draw $\text{co}$ edges from $\text{init}$ to other writes (as it is trivially $\text{co}$-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 $\text{porf} \triangleq (\text{po} \cup r_f)^+\text{prefix}$). For instance, in the right-to-left exploration of $w+r+r$, if $W(x, 1)$ revisits the read of $T_3$, the resulting graph does not have the read of $T_2$ because it was added after $T_3$ and is not $\text{porf}$-before $W(x, 1)$.

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

**Example 2** Consider the following variation of $w+r+r$.

$$
\begin{align*}
T_1: & \quad x := 1 \quad \parallel \quad T_2: & \quad r_2 := x \quad \parallel \quad T_3: & \quad x := 2
\end{align*}
$$

Adding the events from left to right, observe that there are two subexplorations where $W(x, 2)$ has the chance to revisit the read of $T_2$: when the latter reads 0 and when it reads 1. These subexplorations are shown in Fig. 1. If $W(x, 2)$ performs the revisit in both, the exact same graph will be created.

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 $\text{co}$-maximally w.r.t. to the $\text{porf}$-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.

<table>
<thead>
<tr>
<th>T1: ( a := y )</th>
<th>T2: if ((x = 1)) (y := 1)</th>
<th>T3: ( x := 2 ) (y := 1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( R(y) \rightarrow R(x) \rightarrow W(x, 1) \rightarrow W(y, 1) )</td>
<td>( R(y) \rightarrow R(x) \rightarrow W(x, 1) \rightarrow W(y, 1) )</td>
<td>( R(y) \rightarrow R(x) \rightarrow W(x, 1) \rightarrow W(y, 1) )</td>
</tr>
</tbody>
</table>

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. 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., \( \not\exists w' \in E. \ (w, w') \in co \)).

---

1These 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., \( \langle e', w \rangle \in \text{porf} \)):

- If \( e \in W \), then \( e \) is co-maximal in \( E \).
- If \( e \in R \), then \( G.rf(e) \) 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 \( T_2 \) and \( T_3 \) share their code.

\[
T_1: x := 1 \parallel T_2: r_2 := x \parallel T_3: r_3 := x \quad (w+r+r)
\]

We say that executions 2 and 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 2 is the representative one, as in graph 3 the read of \( T_2 \) reads a value that is co-after the one read by \( T_3 \).

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: eco \( \triangleq (\text{co} \cup \text{rf} \cup \text{rb})^+ \), where \( \text{rb} \triangleq \text{rf}^{-1} \); 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 2 in Example 1 is the representative because eco agrees with the thread order (there is an rb; rf path from \( T_2 \) to \( T_3 \)), but graph 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.

\[ ^2 \text{Recall that all writes are co-after the initializer event.} \]
Example 4 To illustrate the problem, consider the following variation of $w+r+r$ (again, $T_2$ and $T_3$ share their code), and suppose we are interested in the executions where $a = 1$.

$$\begin{align*}
T_1: & x := 1 \quad T_2: r_2 := x \\
a := y \quad T_3: r_3 := x \\
T_4: & y := 1
\end{align*}$$

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.

Suppose DPOR first adds the read of $T_3$, and then proceeds with the events of $T_1$. 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 $T_3$ 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 $T_3$, in all subsequent subexplorations $T_3$ keep reading 1; see Prop. 2.2.)

At the next step, the algorithm will add the read of $T_2$, which can either read 1 (from $T_1$) 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 $T_2$ reads 0 and $T_3$ reads 1 is.)

At the final step, the algorithm will add the $W(y, 1)$ event of $T_4$, 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 $T_2$ 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 $T_3$ was added before $T_2$ (i.e., against thread order), and $T_2$ had co-later options available to it ($T_1$ was added after $T_3$ but before $T_2$). 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 $T_2$ and $T_3$ share their code, along with one of its representative executions.

\[
\begin{align*}
T_1: & \quad a := y \\
T_2: & \quad r_2 := x \\
& \quad x := 1 \\
& \quad y := 1 \\
T_3: & \quad r_3 := x \\
& \quad x := 1 \\
& \quad y := 1
\end{align*}
\]

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 $T_3$. 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 $T_3$ is added to K, there are three options:

**L**: $W(x, 1)$ is added co-after $T_2$’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 $T_3$, it will be unable to revisit $R(y)$ because the $W(x, 1)$ of $T_2$ is not maximally added w.r.t. $T_3$’s $W(y, 1)$: it is co-before $T_3$’s $W(x, 1)$, which is in $T_3$’s porf-prefix.
\[ M \]: \( W(x, 1) \) is \( \text{co} \)-before \( T_2 \)'s \( W(x, 1) \). This execution is dropped because \( \text{co} \) contradicts thread-order of symmetric events.

\[ N \]: \( W(x, 1) \) revisits the \( R(x) \) of \( T_2 \). This execution is also dropped because it is not a representative one (\( T_2 \) is reading a \( \text{co} \)-earlier value than \( T_3 \)).

As the \( \text{R+RWW+RWW} \) 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 \( \text{L} \)). In turn, in the representative execution \( \text{L} \), a \( \text{co} \)-edge from a symmetric thread to the \( \text{porf} \)-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, \text{Spore}'s solution is to consider symmetric events part of the revisiting write's prefix. In the case of \( \text{R+RWW+RWW} \), when \text{Spore} considers the revisit between the \( W(y, 1) \) of \( T_3 \) and the \( R(y) \) of \( T_1 \), the \text{prefix} of \( W(y, 1) \) will include not just the events \( \text{porf\-before} \) it, but also the \text{porf} \-prefix of symmetric events as well (namely, event \( W(x, 1) \) of \( T_2 \)). As such, graph \( 42 \) will be generated from \( \text{L} \) because all the affected events (namely, \( T_1 \)'s \( R(y) \) and \( T_2 \)'s \( W(y, 1) \)) are added maximally w.r.t. the \text{new} \text{prefix} of \( W(y, 1) \).

2.2.4 Problem #3: Handling \( \text{po} \cup \text{rf} \cup \text{co} \) 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 \text{sprefix}) in \text{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 \text{Spore} incomplete?

The answer depends on the underlying memory model. First, we can show that \text{sprefix} cycles boil down to \( \text{po} \cup \text{rf} \cup \text{co} \) 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 \text{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 \text{sprefix} but not in its own \text{porf} \-prefix. Such a scenario is shown below.

\[ \text{Example 6} \] Consider the \text{sp-cyc} program, where \( T_2 \) and \( T_3 \) share their code.

\[ T_1: x := 2 \begin{array}{l}
T_2: r_2 := x \quad \text{if} \ (r_2 = 2) \quad y := 1
\end{array}
\begin{array}{l}
T_3: r_3 := x \quad \text{if} \ (r_3 = 2) \quad y := 1
\end{array}
\begin{array}{l}
T_4: a := y \quad x := 1
\end{array}
\]

In the execution of Example 6, \( W(x, 1) \) is in its own \text{sprefix} (\( W(x, 1) \) is read from the \( R(x) \) of \( T_2 \), which is symmetric to the \( R(x) \) of \( T_3 \), which is in turn in the prefix of \( W(x, 1) \)), but not in its own \text{porf} \-prefix (there is no \text{porf} \ cycle).

To restore completeness, \text{Spore} therefore checks that no consistent execution graph has a \( \text{po} \cup \text{rf} \cup \text{co} \) cycle. This condition typically holds: a \( \text{po} \cup \text{rf} \cup \text{co} \) cycle implies that there exist two writes that are not \text{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, \text{Spore} is directly applicable to realistic libraries of concurrent data structures.
enqueuer(v)  iff
node := malloc(...)
node.value := v
node.next := NULL
do
  t := tail
  next := t.next
  if (t != tail) continue if (next != NULL) 
    CAS(tail, t, next)
  continue
  while (~CAS(t.next, next, node))
    CAS(tail, t, node)
dequeuer()
  do
    h := head
    n := h.next
    if (h != head) continue if (n != NULL) return None
  while (~CAS(head, h, n))
    t := tail
    if (h = t)
    CAS(tail, t, n)
    v := n.value
    reclaim(h)
  return v
rdcss_read(a2)  iff
r := a2
while (is_desc(r))
  complete(r)
  r := a2
rdcss(d)  iff
r := CAS(d.a2, d.o2, d)
while (is_desc(r))
  complete(r)
  r := CAS(d.a2, d.o2, 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 enqueuer 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 != NULL), it tries to advance tail on behalf of an incomplete enqueuer.

RDCSS is a double CAS operation that takes as an argument a descriptor d containing two addresses a1, a2 with their expected values o1, o2 and a new value n2. If both addresses contain their expected values, then the new value n2 is stored at the second address a2. To perform the double comparison atomically, RDCSS first tries to place its descriptor in the a2 address, and then reads a1 to determine whether to replace it with the new value n2 or restore the old value o2. 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 in widely used concurrent libraries, including libcdds [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 main and 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 \textit{main} operation succeeds. To this end, Spore changes the underlying memory model and treats \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 \textit{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.

\[\text{Example 7} \] Consider the \textsc{helper-cf} program, along with one of its execution graphs.

\[
\begin{align*}
T_1: & \quad a := \text{CAS}(x, 0, 1) \quad \text{assert}(a = 0) & T_2: & \quad \text{CAS}(x, 0, 1) \\
& & & \text{(HELPER-CF)}
\end{align*}
\]

\(f_m\) 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.

\[\text{Example 8} \] Consider the \textsc{helper-sync} program under SC.

\[
\begin{align*}
T_1: & \quad a := x \quad \text{if } (a = 1) \quad \text{assert}(y = 1) & T_2: & \quad y := 1 & T_3: & \quad \text{CAS}(x, 0, 1) \\
& & & \text{(HELPER-SYNC)} & & \text{Error}
\end{align*}
\]
If the CAS in $T_2$ succeeds and $T_1$’s read of $x$ reads from it, then $T_1$ will necessarily read $y = 1$. If, however, the CAS in $T_3$ succeeds and $T_1$ reads from it (as shown in the graph above), $T_1$ 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, $e \in \text{Event}$, is either the initialization event $\text{init}$, or a thread event $\langle t, i, l \rangle$ where $t \in \text{Tid}$ is a thread identifier, $i \in \text{Idx}$ is a serial number (denoting the index of an event within a thread), and $l \in \text{Lab}$ is a label that takes (at least) one of the following forms:

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

Read and write attributes include the exclusivity flag $\text{excl}$ 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 $G \in \text{EXEC}$ comprises the following components:

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

which in the case of functions annotated with help, comprises three components: (a) a consistency predicate $\text{consistent}(G)$ that determines whether an execution graph $G$ is consistent, and (c) an $\text{IsErroneous}(G)$ 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 $[P]_M$ (resp. $[P]_M^{\text{Annot}}$), is given by the set of execution graphs in $[P]$ (resp. $[P]_M^{\text{Annot}}$) that are $M$-consistent.

In Spore, we assume an underlying memory model $M$ with $\text{cb}_M = (\text{po} \cup \text{rf})^*$, consistent$_M(\cdot)$ being extensible, prefix-closed, and implying RMW atomicity and $\text{cb}_M$-acyclicity [Kokologiannakis et al. 2022], and $\text{IsErroneous}_M(\cdot)$ 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, $\text{SYM}$, with

\[ \text{cb}_\text{SYM} = (\text{po} \cup \text{rf} \cup \text{symb})^* \]

\[ \text{consistent}_\text{SYM}(G) \triangleq \text{consistent}_M(G) \land \text{IRREFLEXIVE}(\text{symb}; \text{eco}) \]

\[ \text{IsErroneous}_\text{SYM}(G) \triangleq \text{IsErroneous}_M(G) \lor \lnot \text{IRREFLEXIVE}((\text{po} \cup \text{rf} \cup \text{co})^*) \]

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

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

\[ \text{po} \triangleq \{ \langle \text{init}, e \rangle \mid e \in \text{Event} \setminus \{\text{init}\} \} \cup \{ \langle t_1, i_1, i_2, t_2, i_2, l_2 \rangle \mid t_1 = t_2 \land i_1 < i_2 \} \]

In our technical appendix [Kokologiannakis et al. 2024b], we define two mappings from programs to sets of execution graphs: (1) $[\cdot]$, which ignores function annotation labels, and simply generates an event with a $M^\text{init}$ label before the events corresponding to the function body; and (2) $[\cdot]^{\text{Annot}}$, which in the case of functions annotated with help, generates only the $M^\text{help}$ event and does not generate any events for the body of the function call. Both mappings keep the $\text{rf}$ and $\text{co}$ 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, $\text{cb}_M$, (b) a consistency predicate $\text{consistent}_M(G)$ that determines whether an execution graph $G$ is consistent, and (c) an $\text{IsErroneous}_M(G)$ 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 $[P]_M$ (resp. $[P]_M^{\text{Annot}}$), is given by the set of execution graphs in $[P]$ (resp. $[P]_M^{\text{Annot}}$) that are $M$-consistent.

In Spore, we assume an underlying memory model $M$ with $\text{cb}_M = (\text{po} \cup \text{rf})^*$, consistent$_M(\cdot)$ being extensible, prefix-closed, and implying RMW atomicity and $\text{cb}_M$-acyclicity [Kokologiannakis et al. 2022], and $\text{IsErroneous}_M(\cdot)$ 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, $\text{SYM}$, with

\[ \text{cb}_\text{SYM} = (\text{po} \cup \text{rf} \cup \text{symb})^* \]

\[ \text{consistent}_\text{SYM}(G) \triangleq \text{consistent}_M(G) \land \text{IRREFLEXIVE}(\text{symb}; \text{eco}) \]

\[ \text{IsErroneous}_\text{SYM}(G) \triangleq \text{IsErroneous}_M(G) \lor \lnot \text{IRREFLEXIVE}((\text{po} \cup \text{rf} \cup \text{co})^*) \]

where $G.\text{symb}$ is the symmetry-before order that orders prefix-matching events according to their thread order. Concretely, $\langle e_1, e_2 \rangle \in G.\text{symb}$ if the following hold:
Definition 3.3 (Annotation correctness). An execution $G$ is correctly annotated if for all $f_h \in G.M_{\text{help}}$, there exist (a) a corresponding $f_m \in G.M_{\text{main}}$ 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 have 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 $\text{loc}/\text{exp}$ to return the location/expected-value of the CAS read following an $f_m \in G.M_{\text{main}}$.

Our definition uses the notion of a source write $s$ at location $\text{loc}(f_m)$, 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 $\text{exp}(f_m)$. 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 $\text{loc}(f_m)$ 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 $f_h \in G.M_{\text{help}}$, there exist (a) a corresponding $f_m \in G.M_{\text{main}}$ with $\text{arg}(f_m) = \text{arg}(f_h)$ and (b) a source write $s \in G.W$ with $\text{loc}(s) = \text{loc}(f_m)$ and $\text{val}(s) = \text{exp}(f_m)$ such that:

- $\langle s, f_m \rangle \in G.rf^2; \text{po}$ (s is observed before $f_m$),
- $\langle f_m, f_h \rangle \in \text{po}^{-1}\text{lim}; G.rf; \text{po}$ (the immediate predecessor of $f_m$ is observed before $f_h$),
- for all $w \in \text{rng}([s]; \text{co})$, $w \in W_{\text{excl}}$ and $\text{val}(w) \neq \text{val}(s)$ (all subsequent writes to $\text{loc}(f_m)$ 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 $G_0$ containing only the initialization event $\text{init}$.

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 po $\cup rf \cup co$ cycles. Such a check is necessary to justify why exploring $\text{cb}_{\text{SYM}}$-acyclic execution graphs suffices: any (po $\cup rf \cup co$)-acyclic graph where the symmetry-before order does not contradict the eco order is also $\text{cb}_{\text{SYM}}$-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\(_{\text{SYM}}(G)\) then exit(“Error”)
3: \(a \leftarrow \text{AddNextEvent}\(_P(G)\)\)
4: if \(a \in R\) then
5: for \(w \in G.W_{\text{loc}}(a)\) do ExploreIfConsistent\(_P(\text{SetRF}(G, a, w))\)
6: else if \(a \in W\) then
7: Explore\(_{\text{COs}}(G, a)\)
8: for \(r \in G.R_{\text{loc}}(a)\) such that \(\langle r, a \rangle \notin G.\text{cb}_{\text{SYM}}\) do
9: \(\text{Deleted} \leftarrow \{e \in G.E \mid r <_G e <_G a \land \langle e, a \rangle \notin G.\text{cb}_{\text{SYM}}\}\)
10: if ShouldRevisit\(_G(\langle r, a, \text{Deleted} \rangle)\) then
11: Explore\(_{\text{COs}}(\text{SetRF}(G \setminus \text{Deleted}, r, a), a)\)
12: else if \(a \neq \bot\) then
13: Explore\(_P(G)\)

14: procedure ExploreIfConsistent\(_P(G)\)
15: if consistent\(_{\text{SYM}}(G)\) then Explore\(_P(G)\)

16: procedure Explore\(_{\text{COs}}(G, a)\)
17: for \(w_p \in G.W_{\text{loc}}(a)\) do ExploreIfConsistent\(_P(\text{SetCO}(G, w_p, a))\)

If the graph is error-free, Explore extends it by one event \(a\) from the program by calling \text{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 \(rf\) 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 \(co\) by means of Explore\(_{\text{COs}}\) (Line 7).

For the revisit case, Explore 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 \(rf\) 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 Explore\(_{\text{COs}}\) 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 \(\text{cb}_{\text{SYM}}\)-acyclic execution graphs. As such, Spore never revisits reads that are in \(\text{cb}_{\text{SYM}}\)-before \(a\) (as opposed to \(\text{cb}_M\)-before \(a\)), as revisiting such reads would create \(\text{cb}_{\text{SYM}}\) cycles (the \(\text{cb}_{\text{SYM}}\)-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 $\mathcal{M}$. Indeed, this is because these are handled implicitly by the interpreter: Line 3 adds events according to our annotated semantics $[P]_{\text{Annot}}$. When the interpreter encounters a function annotated with main, it will yield an $M^\text{main}(as)$ (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 $M^\text{help}(as)$ 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 $\text{cb}_{\text{SYM}}$-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 $[P]_{\text{Annot}}$ (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 $\text{IsErroneous}_M(G)$. Then, there exists $G' \in [P]_M$ such that $\text{IsErroneous}_{\text{SYM}}(G)$.

**Proof Sketch.** It suffices to show that there exists a corresponding execution $G'$ (where every $f_h$ being treated as a (single) dummy event $M^\text{help}(\ldots)$) such that (1) $\text{IsErroneous}_M(G')$ 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 $\pi$ 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 $\pi$ 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 $G \in [P]_{\text{Annot}}$, there is a unique execution graph $G' \in [P]_{\text{Annot}}^{\text{SYM}}$ 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 \texttt{eco}-unordered threads are in fact equal, and that the constructed graph $G'$ satisfies $\text{irreflexive}(\text{symb}; \text{eco})$.

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. $\text{Explore}_P(G_0)$ terminates.
2. $\text{Explore}_P(G_0)$ only explores $\text{cb_SYM}$-prefixes of executions in $[P]_\text{SYM}^{\text{Annot}}$.
3. $\text{Explore}_P(G_0)$ explores every execution $G \in [P]_\text{SYM}^{\text{Annot}}$ such that $\text{irreflexive}(G_.\text{cb_SYM})$.
4. $\text{Explore}_P(G_0)$ 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 $\text{rf/co}$ 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 $[P]_\text{SYM}^{\text{Annot}}$ includes a $\text{cb_SYM}$-cyclic execution, which the algorithm would not explore, then it also includes a $\text{cb_SYM}$-acyclic execution with a $\text{po} \cup \text{rf} \cup \text{co}$ cycle, which the algorithm would explore and report.

**Proposition 3.7 ($\text{cb_SYM}$ cycle).** If there is an execution $G \in [P]_\text{SYM}^{\text{Annot}}$ with a $G_.\text{cb_SYM}$ cycle, then there is an execution $G' \in [P]_\text{SYM}^{\text{Annot}}$ such that $\text{irreflexive}(G'.\text{cb_SYM})$ and $G'$ has a $\text{po} \cup \text{rf} \cup \text{co}$ 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 $G \in [P]_\text{SYM}^{\text{Annot}}$ such that $\text{IsErroneous_SYM}(G)$, then $\text{Explore}_P(G_0)$ will report an error. Otherwise, for each $G \in [P]_\text{SYM}^{\text{Annot}}$, $\text{Explore}_P(G_0)$ will explore an execution $G' \in [P]_\text{SYM}^{\text{Annot}}$ 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 $\text{Explore}_P(G_0)$, $G \not\approx G'$.

## 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 $\text{cb_SYM}$, and made GenMC’s scheduler respect $\text{cb_SYM}$ 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 compare 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):** \( \left\lceil \frac{N}{2} \right\rceil \) (resp. \( \left\lfloor \frac{N}{2} \right\rfloor \)) 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 \( \left\lceil \frac{N}{2} \right\rceil \) (resp. \( \left\lfloor \frac{N}{2} \right\rfloor \)) 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...
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 dequeue method, which is why the results are very similar.

\(^7\)These benchmarks only differ in their dequeue method, which is why the results are very similar.

Fig. 3. Data structure benchmarks: Number of executions exposed (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 fol1queue and treiber, we can make observations similar to the ones for the previous benchmarks, albeit a bit toned down. In the case of fol1queue, 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; Rodriguez 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 seen 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 po ∪ 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


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