Effective Lock Handling in Stateless Model Checking

MICHALIS KOKOLOGIANNAKIS, MPI-SWS, Germany
AZALEA RAAD, MPI-SWS, Germany
VIKTOR VAFEIADIS, MPI-SWS, Germany

Stateless Model Checking (SMC) is a verification technique for concurrent programs that checks for safety violations by exploring all possible thread interleavings. SMC is usually coupled with Partial Order Reduction (POR), which exploits the independence of instructions to avoid redundant explorations when an equivalent one has already been considered. While effective POR techniques have been developed for many different memory models, they are only able to exploit independence at the *instruction level*, which makes them unsuitable for programs with coarse-grained synchronization mechanisms such as *locks*.

We present a lock-aware POR algorithm, *LAPOR*, that exploits independence at both *instruction* and *critical section levels*. This enables LAPOR to explore exponentially fewer interleavings than the state-of-the-art techniques for programs that use locks conservatively. Our algorithm is sound, complete, and optimal, and can be used for verifying programs under several different memory models. We implement LAPOR in a tool and show that it can be exponentially faster than the state-of-the-art model checkers.

CCS Concepts: • **Theory of computation → Verification by model checking**; • **Software and its engineering** → *Software testing and debugging*.

Additional Key Words and Phrases: Model checking, mutual exclusion locks, weak memory models

ACM Reference Format:
Michalis Kokologiannakis, Azalea Raad, and Viktor Vafeiadis. 2019. Effective Lock Handling in Stateless Model Checking. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 173 (October 2019), 40 pages. https://doi.org/10.1145/3360599

1 INTRODUCTION

Coarse-grained locking (CGL) is the most widespread form of synchronization in concurrent shared-memory programs. It is a fairly straightforward way of making a sequential program "thread-safe" with programming languages, such as Java, even providing syntactic support for synchronizing all the methods operating on a shared object (regardless of whether they access the same fields of the object or not). CGL often performs better than finer-grained schemes in cases of low contention or when "hardware lock elision" is available (e.g., on modern x86 processors). Lock elision enables concurrent execution of lock-protected code regions that do not conflict (e.g., access only disjoint memory locations), by optimistically executing them and rolling back in case a conflict is detected.

Intuitively, CGL simplifies reasoning about concurrent programs because it reduces the number of interleavings that a programmer or an automated verifier has to consider. Somewhat surprisingly, however, CGL has the opposite effect on verification using the state-of-the-art *stateless model checking* (SMC) tools (e.g., [Abdulla et al. 2014, 2018; Aronis et al. 2018; Kokologiannakis et al. 2017,

Authors' addresses: Michalis Kokologiannakis, MPI-SWS, Saarland Informatics Campus, Germany, michalis@mpi-sws.org; Azalea Raad, MPI-SWS, Saarland Informatics Campus, Germany, azalea@mpi-sws.org; Viktor Vafeiadis, MPI-SWS, Saarland Informatics Campus, Germany, viktor@mpi-sws.org.

2475-1421/2019/10-ART173
https://doi.org/10.1145/3360599

2019]). More specifically, CGL increases the verification cost significantly because it inhibits the most important optimization of SMC, which makes the approach scalable, namely *(dynamic) partial order reduction* (POR) [Abdulla et al. 2014; Flanagan and Godefroid 2005].

SMC with POR explores all possible program executions (i.e., thread interleavings under sequential consistency) up to a fixed equivalence relation, where two interleavings are deemed equivalent if one can be obtained from the other by commuting pairs of independent actions (e.g., accesses to different locations). Since accesses of different threads are most often independent, POR leads to a huge reduction in the space that needs to be explored, which in turn makes SMC with POR scalable.

In programs with coarse-grained locks, however, existing POR algorithms achieve very little reduction in the state space because they treat locking commands as normal instructions. Since acquisitions of the same lock cannot be commuted (because it would affect which thread acquires the lock), existing SMC/POR tools consider all the possible ways in which critical sections of the same lock could be interleaved. This leads to a lot of redundant exploration because the relative order of critical sections is often irrelevant—it does not affect the program outcome.

To make SMC scalable for coarse-grained locking programs, we develop the *Lock-Aware Partial Order Reduction* (LAPOR) algorithm, which treats lock acquisitions in each program execution as independent unless their respective order can be deduced from the memory accesses of the execution. As such, although our algorithm works at the instruction level, it also exploits independence at the critical section level, thus achieving exponential reductions in the number of explored executions compared to the state-of-the-art model checking tools.

In more detail, this paper makes the following contributions.

(§2) We present an intuitive account of LAPOR through a series of examples, and show how the lock acquisition order can be inferred on the fly when necessary, without exploring all possible orderings in advance.

(§3) We present the formalism underpinning LAPOR. Our algorithm is parametric both in the choice of the memory model and in when two concurrent writes are deemed conflicting.

(§4) We describe LAPOR in detail, and show that it is sound (produces no false positives), complete (explores all possible behaviours), and optimal (explores each behaviour exactly once), and that it can be applied to any memory model that satisfies a few basic assumptions.

(§5) We implement LAPOR over the state-of-the-art GenMC model checker [Kokologiannakis et al. 2019] and evaluate its performance. We show that LAPOR is significantly faster than the state of the art for programs with coarse-grained locks, while incurring only a small overhead for programs with fine-grained locks, and negligible overhead for programs without locks.

2 OVERVIEW

2.1 Stateless Model Checking and Dynamic Partial Order Reduction

Stateless Model Checking (SMC) is an effective concurrency verification technique based on the observation that one can verify a concurrent program by exploring all possible interleavings of its threads[1]. However, as exploring all interleavings quickly becomes non-scalable even for small programs, *partial order reduction* (POR) techniques are often employed alongside SMC. Given a suitable notion of *instruction independence*, POR techniques deem two interleavings equivalent if one can be obtained from the other by swapping adjacent, independent execution steps. POR techniques then explore (at least) one interleaving from each equivalence class—in the context of SC, these equivalence classes are known as Mazurkiewicz traces [Mazurkiewicz 1987].

[1] Although the approach presented in this paper is parametric within the choice of the memory model (see §3), in this section we assume a setting of sequential consistency (SC) [Lamport 1979] for ease of presentation.

Amongst POR techniques, Dynamic Partial Order Reduction (DPOR) [Flanagan and Godefroid 2005] stands out. Unlike earlier POR techniques, DPOR does not rely on static over-approximations to infer whether events are independent. It instead keeps track of the conflicting events along the exploration path, and uses them to guide further explorations.

To make this concrete, consider the following program with N concurrent reads.[2]

$$a_1 := x \;\|\; \cdots \;\|\; a_N := x \tag{N-READS}$$

While this program has $N!$ interleavings, almost all (D)POR techniques will explore only one interleaving by detecting that the read operations are independent (they return the same value irrespective of their ordering) and hence all interleavings are deemed equivalent.

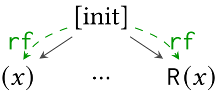


Fig. 1. Execution graph.

Each equivalence class of interleavings can be represented as an *execution graph*, such as the one shown in Fig. 1 for the N-READS program. The nodes of an execution graph denote execution *events* which correspond to program instructions, while its edges describe different relations on events. More specifically, the solid black edges represent the *program-order* (po) relation, which totally orders the events in each thread and orders the initialization events before all other events. The dashed edges represent the *reads-from* (rf) relation, which relates each write event to the read events that read from it and obtain its value. rf is total and functional on its second argument: every reads reads from precisely one write. In the graph of Fig. 1, all read events read from the initialization event. Note that the reads are not related to one another as they are independent.

The objective of SMC is to explore all execution graphs of a given program in a systematic way, while keeping only one graph in memory at any given time. In the remainder of this article we use the terms 'execution graphs', 'executions', 'graphs' and 'equivalence classes' interchangeably.

2.2 Locks in Stateless Model Checking

Although the notion of instruction independence is well-studied for memory accesses (reads and writes), exploiting independence is much more difficult in the presence of locks. To see why, consider a variant of the N-READS program where each read is wrapped in a critical section using locks:

$$\begin{array}{l} \mathit{lock}(l); \\ a_1 := x; \\ \mathit{unlock}(l) \end{array} \;\Bigg\|\; \cdots \;\Bigg\|\; \begin{array}{l} \mathit{lock}(l); \\ a_N := x; \\ \mathit{unlock}(l) \end{array} \tag{N-READS-LOCK}$$

For such locking programs, existing (D)POR techniques take an either *fine-grained* or *coarse-grained* approach to locks.

Fine-grained Approaches. Fine-grained approaches consider interleavings at the instruction level, treating $\mathit{lock}()$ and $\mathit{unlock}()$ operations as individual instructions. In the case of N-READS-LOCK, the lock operations are considered as dependent because they access the same lock and cannot be reordered. This yields $N!$ distinct equivalence classes capturing the order in which the lock is acquired. That is, existing (D)POR techniques explore $N!$ interleavings (one per execution) instead of one as in the case of N-READS.

More concretely, for $N = 2$, assume without loss of generality that a (D)POR algorithm first obtains execution Ⓐ in Fig. 2, where the left thread acquires the lock first. The algorithm subsequently detects that the two threads compete to obtain the same lock; it thus reverses the order in which the lock is acquired, yielding execution Ⓑ. As such, although adding locks to N-READS does not alter the program behaviour, it results in additional *unnecessary* executions.

[2] In our examples, we use a, b, c for local variables, and x, y, z for global (shared) variables, which are initialized with 0.

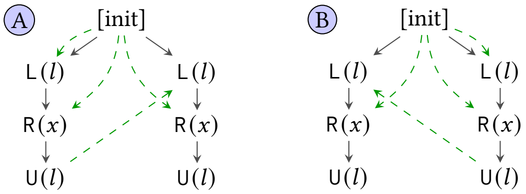


Fig. 2. Executions of the N-reads-lock program for $N = 2$ under a classic POR algorithm.

The reason for this is that traditional (D)POR algorithms treat *lock*() and *unlock*() operations as memory accesses (reads and writes). More specifically, *lock*() is assigned read semantics, *unlock*() is assigned write semantics, and `rf` is required to be one-to-one on lock/unlock events. This characterization of locks captures the order of lock acquisition elegantly: when the lock operation of a critical section cs_2 reads from the unlock operation in critical section cs_1, then cs_1 is ordered before cs_2. That is, the `rf` relation induces a total order on critical sections and thus determines the lock acquisition order. As lock operations participate in `rf`, SMC algorithms always consider the lock operations as dependent. As we have seen, this can then lead to an exponential increase in the number of executions, hindering performance needlessly.

This can be very surprising to programmers using SMC to test their code because it contradicts the intuition that the behaviour of concurrent programs is *monotonic* with respect to synchronization: adding more synchronization (e.g., locks) to a program should not introduce additional behaviours. As such, when inserting locks in the N-reads program to obtain N-reads-lock, the possible behaviours of N-reads-lock must be included in those of N-reads. As SMC techniques are used to explore the possible behaviours of a program by considering all its executions, under the monotonicity intuition one then does not expect N-reads-lock to have more executions than N-reads. However, while recent SMC techniques generate only one execution for N-reads (ignoring the read order), they generate $N!$ executions for the N-reads-lock variant (considering all lock orderings), thus breaking the monotonicity property.

Coarse-grained Approaches. In coarse-grained approaches, (D)POR techniques treat critical sections as *atomic* blocks, and only consider interleavings at the level of critical sections. As such, for the N-reads-lock above, these techniques correctly identify the $N!$ possible interleavings of the critical sections as equivalent, thus generating one execution. Such atomic treatment of critical sections may improve performance significantly in *conservative* settings where all accesses to shared variables are enclosed within critical sections (e.g., N-reads-lock). However, in *mixed* settings where shared variables may be accessed both within and without critical sections, such atomic treatment is *unsound* as it fails to explore all executions.

To see this, consider the conservative program below (left) and its mixed variant (right):

$$\begin{array}{l} lock(l);\\ x := 1;\\ x := 2;\\ unlock(l) \end{array} \;\Bigg\|\; \begin{array}{l} lock(l);\\ a := x;\\ unlock(l) \end{array} \quad (\text{ww+r-cons}) \qquad\qquad \begin{array}{l} lock(l);\\ x := 1;\\ x := 2;\\ unlock(l) \end{array} \;\Bigg\|\; a := x \quad (\text{ww+r-mixed})$$

The ww+r-cons program has two possible executions: one where the left critical section is executed first and thus $a = 2$, another where the right critical section is executed first and thus $a = 0$. The ww+r-mixed program also has a third possible execution. Since the right thread does not acquire the lock l, the critical section in the left thread need not execute atomically with respect to the right thread, and so the read may interleave between the two writes yielding the outcome $a = 1$.

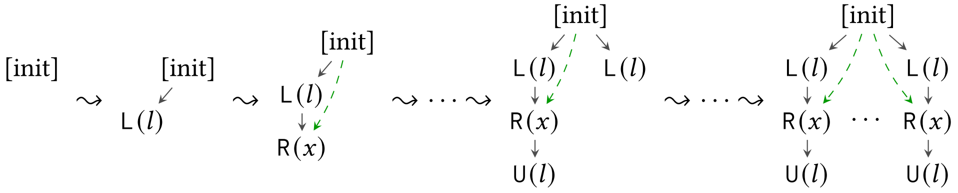


Fig. 3. Full exploration for the N-reads-lock program.

When treating critical sections atomically, (D)POR techniques cannot distinguish between ww+r-cons and its mixed variant ww+r-mixed, failing to explore the valid execution in which $a = 1$. As such, atomic treatment of critical sections is unsound in the mixed setting.

2.3 LAPOR: Keeping Locks Unordered

LAPOR follows the fine-grained approach of considering interleavings at the level of instructions. It therefore records each individual instruction as an event in an execution. In order to avoid redundant executions due to locks, however, unlike classical (D)POR approaches, LAPOR does not assign read/write semantics to locks; nor does it record the total lock acquisition order. Instead, LAPOR keeps the lock operations *unordered*, and infers the lock order on the fly, only *when necessary*.

Let us demonstrate LAPOR via the N-reads-lock example. As shown in Fig. 3, starting from an empty execution (containing only the initialization events), LAPOR inspects the program and adds execution events one at a time, thus arriving at a full execution where the locks are unordered. Since there is only one reads-from option (i.e., the initialization event) for the reads in each thread, and since locks do not participate in `rf`, LAPOR explores *only one* execution rather than $N!$ executions.

2.4 LAPOR: Inferring Lock Orderings On the Fly

In the N-reads-lock program above, the order of lock acquisition does not matter as the locks do not alter the behaviour of the program. This, however, is not always the case because locks may induce mutual exclusion constraints on the execution. In such cases, LAPOR *infers* the necessary lock orderings on the fly. To demonstrate this, consider the example below:

```
lock(l);   || lock(l);
x := 42;   || a := x;
y := 42;   || b := y;
unlock(l)  || unlock(l)
```

(ww+rr-lock)

Here, the order of lock acquisition matters because the values read from x and y depend on the order in which the locks are acquired: when the left thread acquires the lock first, the right thread reads 42 for x and y; when the right thread acquires the lock first, the right thread reads 0 (the initialization value) for x and y. Let us describe how LAPOR infers the lock ordering in ww+rr-lock.

As before, starting from the empty graph (containing initialization events only), LAPOR incrementally adds all events of the first thread, as well as the lock event of the second thread, arriving at execution ① of Fig. 4. LAPOR then proceeds to add the event associated with the $a := x$ read. Inspecting the execution so far, LAPOR notes that this read may read either 0 or 42, and thus both possibilities must be explored, as depicted in executions ② and ③ of Fig. 4. However, rather than exploring both executions at once, LAPOR proceeds with one of the executions (e.g., ②), and pushes the alternative execution into a *work set* to be explored at a later time.

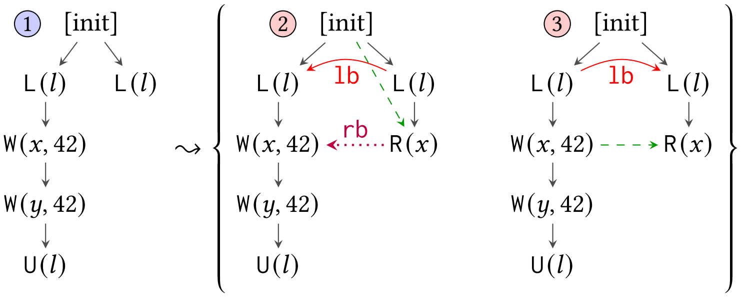


Fig. 4. Key exploration step for WW+RR-LOCK.

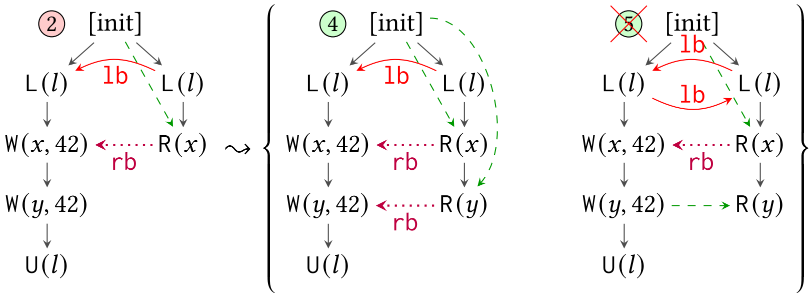


Fig. 5. Key exploration step for WW+RR-LOCK (continued).

Although reading 0 and 42 are both consistent options, reading either value induces a certain lock ordering, as discussed above. That is, the order of lock acquisition matters because the two critical sections are *not independent*. We refer to two such critical sections as *conflicting*. For example, in WW+RR-LOCK, the two critical sections are read-write conflicted because the right thread reads from location x, and the left thread also writes at that memory location. As we describe shortly, LAPOR detects conflicting critical sections, infers their induced lock orderings, and tracks them as an additional relation in the execution, namely the *locks-before* relation: lb.

In the case where $a := x$ reads 0 (in ②), LAPOR proceeds as follows. First, it adds a *reads-before* (a.k.a. from-read) edge (rb) from $a := x$ to $x := 42$, as depicted in ②. Intuitively, rb edges relate each read r (e.g., $a := x$) to all the writes (e.g., $x := 42$) that overwrite the value read by r. As such, the presence of an rb edge between the critical sections of distinct threads indicates a conflict: both critical sections access the same location (e.g., x) and one is overwriting the value read by the other.

Next, LAPOR determines the induced lock ordering as prescribed by rb: when there is an rb edge from critical section cs_1 to cs_2, then cs_1 must have acquired the lock before cs_2, and thus there is an lb edge from cs_1 to cs_2. This is illustrated in ② (Fig. 4) by the lb edge from the critical section in the right thread to that in the left.

Analogously, the presence of an rf edge between the critical sections of distinct threads indicates a conflict: both critical sections access the same location and one reads a value written by the other. As such, the two critical sections in ③ (Fig. 4) are deemed conflicted and LAPOR adds an lb edge from the critical section in the left thread to that in the right.

Continuing from ②, LAPOR next adds the $b := y$ read, which may once again read either 0 or 42, as depicted in graphs ④ and ⑤ of Fig. 5). Note that if locks did not enforce mutual exclusion, both read options would be valid. However, since locks do enforce mutual exclusion, reading 0 is the

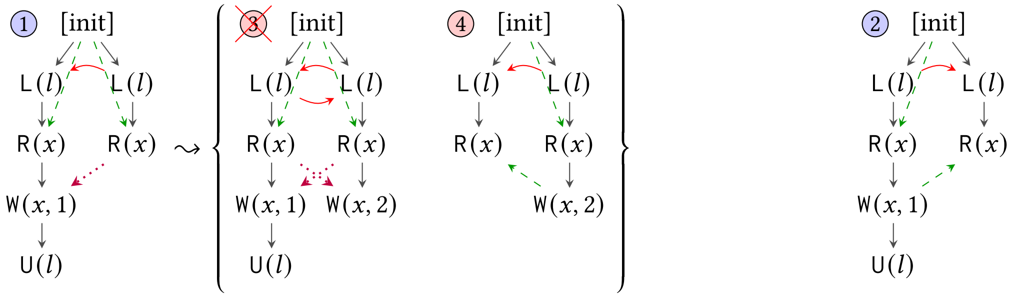


Fig. 6. Key exploration step for 2rw-lock.

only valid option as reading 42 violates mutual exclusion due to the lock ordering imposed by lb. Let us now discuss how LAPOR detects this mutual exclusion violation.

In the case where $b := y$ reads 42, i.e., there is an rf edge from $y := 42$ to $b := y$ as shown in ⑤, LAPOR detects a conflict between the two critical sections, and thus adds an lb edge from the critical section in the left thread to that in the right. This then leads to an lb cycle between the two critical sections, which means that the resulting graph is inconsistent, because a critical section cannot be simultaneously ordered both before and after another one. Thus reading 42 is discarded as invalid. Subsequently, the execution in ④ is completed by adding the unlock event of the right thread, thus concluding the first run of LAPOR.

Lastly, LAPOR completes execution ③ in Fig. 4 following analogous steps: it adds the $b := y$ read, excludes 0 as a possible reads-from value for $b := y$ (otherwise we obtain an lb cycle resulting in an inconsistent execution), and arrives at the execution where both $a := x$ and $b := y$ read 42.

2.5 LAPOR: Excluding Inconsistent Executions

In §2.4 we showed that when adding a read r to the graph, some reads-from options for r may lead to inconsistent executions. Nevertheless, a read can always be safely added to the graph as there is at least one reads-from option that preserves consistency. However, as we demonstrate here, this is not the case when adding a write to the graph; i.e., the mere addition of a write to the graph may render it inconsistent. Consider the example below:

$$\begin{array}{l||l} \mathit{lock}(l); & \mathit{lock}(l); \\ a := x; & b := x; \\ \textbf{if } a = 0 \textbf{ then } x := 1; & x := 2; \\ \mathit{unlock}(l) & \mathit{unlock}(l) \end{array} \qquad (\text{2rw-lock})$$

Starting from the empty graph with initialization events, LAPOR first adds $\mathit{lock}(l)$ and $a := x$ in the left thread. The $a := x$ read can only read from the initialization event since the graph contains no other writes. This makes the condition of the **if** statement true, enabling the addition of $x := 1$. LAPOR then adds $\mathit{unlock}(l)$ in the left thread, as well as $\mathit{lock}(l)$ in the second thread. It next adds the $b := x$ read, detecting that there are two available reads-from options (either form the initialization event, or the $x := 1$ write in the first thread), as depicted in executions ① and ② of Fig. 6.

Observe that as depicted in ① (resp. ②), the choice of reads-from option for $b := x$ results in an rb (resp. rf) edge between $x := 1$ and $b := x$, which in turn induces an order on the two critical sections, as discussed earlier. As before, LAPOR continues with one of the executions (e.g., ①), and the other execution (e.g., ②) is added to a work set for later exploration. Continuing from ①, LAPOR next adds the $x := 2$ write, which brings up two issues that require additional care.

First, the addition of the $x := 2$ *write* results in an rb edge from $a := x$ to $x := 2$, which in turn induces an lb edge from the left critical section to the right, as depicted in ③. This then leads to an lb cycle, rendering the execution inconsistent; as such, graph ③ is not explored further.

Second, as discussed above, previously when $a := x$ was added, there was only one read-from option: the initialization event. However, note that in an interleaving of 2RW-LOCK where the right thread is executed before the left, it is possible for $a := x$ to read from $x := 2$ in the right thread. As such, we must ensure that this alternative interleaving is also explored. To do this, when adding a write w (e.g., $x := 2$), regardless of whether adding w makes the graph inconsistent, we check if w may *revisit* an existing read r in the graph (e.g., $a := x$), in that r may read from the newly added w, leading to an additional execution (e.g., ④ in Fig. 6).

When this is the case, following Abdulla et al. [2018] and Kokologiannakis et al. [2019], we only keep the following events in this additional execution: (1) r itself (e.g., $a := x$), (2) the events that were added *before* r (e.g., $lock(l)$ in the left thread), (3) w itself (e.g., $x := 2$), and (4) the events that led to w (e.g., $lock(l)$ and $b := x$ in the right thread). In other words, we remove all events that were added *after* r but did not lead to w (e.g., the $x := 1$ and $unlock(l)$ in the left thread). This is because the new value read by r (from w) may affect the control flow, and lead to different events in the graph. For example, in this new execution where $x := 2$ revisits $a := x$ (i.e., $a := x$ reads from $x := 2$), the condition of the **if** statement is no longer true, and thus $x := 1$ is no longer part of the execution, and must be removed from the graph, as depicted in ④.

Once again, the new execution ④ is added to the work set for later exploration, and thus the work set contains ② and ④ at this point. As the current execution ③ was dropped due to inconsistency, LAPOR proceeds by exploring the remaining executions in the work set.

Exploring ② is straightforward and is simply completed by adding the remaining events of the right thread, yielding an execution where $a = 0$ and $b = 1$. Analogously, continuing from ④, the $x := 1$ write is no longer added (as the conditional fails), and the exploration is completed by adding the $unlock(l)$ event of the left thread, yielding an execution where $a = 2$ and $b = 0$.

Remark 1. Note that when adding $b := x$ reading 0 (from the initialization event) in graph ③, LAPOR cannot know in advance that this leads to an inconsistent execution at the next step (after adding $x := 2$). This is because LAPOR does not know of the next events that are added to the graph in due course. In particular, if $b := x$ were the only event in the critical section of the right thread, reading 0 would be a valid option and would not lead to an inconsistent execution. However, were we to disallow $b := x$ reading 0 (fearing that it may lead to inconsistency), we would inadvertently miss the *consistent* execution in which both $a := x$ and $b := x$ read 0 from the initialization event.

2.6 LAPOR: Parametric Treatment of Write-Write Conflicts

In the previous sections we showed how LAPOR uses rb and rf to detect *read-write* conflicts, i.e., when one critical section writes to a location read by the other. We next describe how LAPOR deals with *write-write* conflicts, i.e., when two critical sections write to the same location without reading from it. Consider the W+W+LOCK example below (left), whose two critical sections are conflicting because they both access x and at least one of them is a write.

$$\begin{array}{l||l} lock(l); & lock(l); \\ x := 1; & x := 2; \\ unlock(l) & unlock(l) \end{array} \qquad (\text{W+W+LOCK}) \qquad\qquad x := 1 \;||\; x := 2 \qquad (\text{W+W})$$

The W+W+LOCK program has two interleavings: one where the left thread acquires the lock first, another where the right thread acquires the lock first. The order in which the two critical sections execute determines the final value of x, but does not affect any observation made by the program (since the program does not read x). As such, it suffices for a model checker to explore only one of

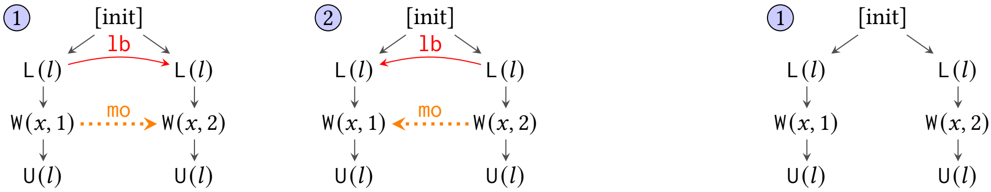


Fig. 7. Executions of the W+W+LOCK program via mo (left) and wb (right)

the two interleavings. LAPOR can achieve this, but it depends on how exactly it is instantiated to order concurrent writes. As we will shortly see, LAPOR is parametric in its treatment of write-write conflicts and will explore one or two interleavings depending on the underlying DPOR algorithm used for ordering normal memory accesses.

To explain this, let us look at how existing DPOR algorithms handle the W+W example above (right), obtained from W+W+LOCK by removing the lock operations.

Classical DPOR algorithms [Abdulla et al. 2014; Flanagan and Godefroid 2005] consider the two writes on x conflicting, and thus explore two interleavings: one where $x := 1$ executes first and another where $x := 2$ executes first. To distinguish between the two interleavings, execution graphs in classical DPOR algorithms (in addition to po and rf) record the '*modification-order*', mo, which totally orders the writes on each location. Execution graphs recording mo are often called Mazurkiewicz traces [Mazurkiewicz 1987] or Shasha-Snir traces [Shasha and Snir 1988] in the literature. LAPOR can be instantiated with such execution graphs by also using mo to infer the induced lock ordering in the same way as with rf and rb. For the W+W+LOCK program, this leads to two executions as depicted in Fig. 7 (left).

However, several more recent DPOR algorithms [Abdulla et al. 2018; Aronis et al. 2018; Chalupa et al. 2017; Kokologiannakis et al. 2017, 2019] do not consider all orderings on the writes of the same location *unless* this order is *observable*. As such, since the value of x is not read in W+W, the order of the writes cannot possibly be observable, and thus these techniques explore only one execution for W+W, ignoring the order in which the writes are executed.

To do this, several such techniques compute the '*writes-before*' relation, wb ⊆ mo, which (similarly to lb) *infers* the order between writes. That is, wb orders writes on the same location only when necessary. For example, consider the execution of Fig. 8. In this execution, W$(x, 2)$ must *write before* W$(x, 3)$ as they are po-related. Moreover, W$(x, 1)$ must write before W$(x, 3)$: otherwise, R(x) would read 1 because of coherence. However, observe that W$(x, 2)$ and W$(x, 1)$ are not ordered. Intuitively, so long as wb is respected for x, these can execute in either order as this order cannot be observed.

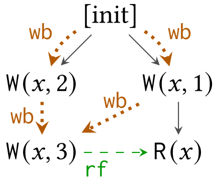


Fig. 8. The wb relation

LAPOR can be instantiated with such techniques by using wb instead of mo to induce the lock orderings between critical sections with write-write conflicts. For the W+W+LOCK example above, the order between the two writes is unnecessary; thus the recorded wb relation is empty, leading to a single execution as in Fig. 7 (right).

This parametricity of LAPOR is a significant part of our contribution: we reduce the number of explorations exponentially by recording the lock ordering when necessary by (1) identifying read-write conflicted critical sections, and (2) taking advantage of the underlying (D)POR technique and utilizing its notion of conflicts to detect write-write-conflicted critical sections.

3 FORMAL MODEL

We present the formalism underpinning LAPOR.

Labels, Events and Executions. As discussed in §2, in the literature of declarative (a.k.a. axiomatic) concurrency models, the traces of a program are typically represented as a set of execution graphs, where the graph nodes denote execution *events*, and graph edges capture the sundry relations on events. Each event corresponds to the execution of an instruction. An event is a tuple of the form $\langle i, n, l\rangle$, where $i \in \mathsf{Tid} \uplus \{0\}$ is a *thread identifier* (0 for initialization events) with $\mathsf{Tid} \subseteq \mathbb{N}$, $n \in \mathbb{N}$ is the *serial number* inside a thread, and $l \in \{\mathtt{error}\} \uplus \mathsf{Lab}$ is an event *label.* The serial number of an event denotes its index (from 1) within its thread; e.g., the first event of a thread has serial number 1. Serial number 0 is reserved for initialization events.

A label may be either: (i) the *error* label $\mathtt{error}$ (denoting assertion violations); or (ii) an *instruction* label $l \in \mathsf{Lab}$. As the set of instructions is memory model (MM) specific, the set of event labels is also MM-specific. As such, we keep our definitions parametric in the choice of MM, and assume a set of *labels*, Lab, associated with the instructions of the underlying MM. For instance, under the SC memory model, the store instruction $x := 1$ is associated with the label $\mathtt{W}(x, 1)$. To model instruction labels, we assume a set of memory locations, Loc, ranged over by x, y, z. We further assume a set of *read labels*, $\mathsf{RLab} \subseteq \mathsf{Lab}$, a set of *write labels*, $\mathsf{WLab} \subseteq \mathsf{Lab}$, a set of *lock labels*, $\mathsf{LLab} \subseteq \mathsf{Lab}$, and a set of *unlock labels*, $\mathsf{ULab} \subseteq \mathsf{Lab}$, associated with *read* (load), *write* (store), *lock* and *unlock* instructions, respectively. For instance, the SC label $\mathtt{W}(x, 1)$ is a write label.

Definition 3.1 (Events). Given a set of labels Lab, an *event* $e \in \mathsf{Event}$ is a tuple $\langle i, n, l\rangle$, where $i \in \mathsf{Tid}$ is a thread identifier, $n \in \mathbb{N}$ is a sequence number, and $l \in \{\mathtt{error}\} \uplus \mathsf{Lab}$ is a label.

We typically use a, b and e to range over events. The functions $\mathtt{tid}$, $\mathtt{sn}$, and $\mathtt{lab}$ project the thread identifier, the sequence number, and the label of an event respectively. We assume a set of locations Loc, and assume that functions $\mathtt{loc}$, $\mathtt{val}_\mathtt{r}$ and $\mathtt{val}_\mathtt{w}$ respectively project the location, the read value and the written value of a label, where applicable. For instance, $\mathtt{loc}(l)=x$ and $\mathtt{val}_\mathtt{w}(l)=1$ for $l=\mathtt{W}(x, 1)$. We lift the label functions $\mathtt{loc}$, $\mathtt{val}_\mathtt{r}$ and $\mathtt{val}_\mathtt{w}$ to events, and write e.g., $\mathtt{loc}(e)$ for $\mathtt{loc}(\mathtt{lab}(e))$, given an event e. We define the *read events* as $\mathtt{R} \triangleq \{e \in \mathsf{Event} \mid \mathtt{lab}(e) \in \mathsf{RLab}\}$; the write ($\mathtt{W}$), lock ($\mathtt{L}$) and unlock ($\mathtt{U}$) events are defined analogously. We define *initialization events* as $\mathsf{Event}_0 \triangleq \{e \in \mathtt{W} \mid \mathtt{tid}(e)=0\}$.

Definition 3.2 (Executions). An *execution* is a tuple $G=\langle E, \mathit{rf}\rangle$, where $E \subseteq \mathsf{Event}$ is a sequence of *events*, and $\mathit{rf} \subseteq (E \cap \mathtt{W}) \times (E \cap \mathtt{R})$ is the *reads-from* relation, which is total and functional on its second argument: every read event reads from exactly one write event.

The order of events in the sequence E denotes the order in which the events were added to G. Given an execution G, we write $G.\mathsf{E}$ and $G.\mathtt{rf}$ for its components, and write $G.\mathsf{R}$ for $G.\mathsf{E} \cap \mathsf{R}$; similarly for $G.\mathtt{W}$, $G.\mathsf{L}$ and $G.\mathsf{U}$. We write $G.\mathsf{E}_0$ for $G.\mathsf{E} \cap \mathsf{Event}_0$; we write $G.\mathsf{E}_i$ for $\{\langle i', -, -\rangle \in G.\mathsf{E} \mid i = i'\}$; and write $G.\mathsf{po}$ for the *program order* defined as follows:

$$G.\mathsf{po} \triangleq G.\mathsf{E}_0 \times (G.\mathsf{E} \setminus G.\mathsf{E}_0) \cup \left\{ \begin{array}{l|l} \langle\langle i_1, n_1, l_1\rangle, & \langle i_1, n_1, l_1\rangle, \langle i_2, n_2, l_2\rangle \in G.\mathsf{E} \setminus G.\mathsf{E}_0 \\ \quad \langle i_2, n_2, l_2\rangle\rangle & \wedge\ i_1 = i_2 \wedge n_1 < n_2 \end{array} \right\}$$

Initialization events are po-before all other events, and events of the same thread are ordered according to their sequence number. Further, we define $\mathsf{SetRF}(G, w, r)$ as graph obtained from changing the incoming $\mathtt{rf}$ edge of r to come from w. Formally, $\mathsf{SetRF}(\langle E, \mathit{rf}\rangle, w, r) \triangleq \langle E, \mathit{rf} \setminus E \times \{r\} \cup \{\langle w, r\rangle\}\rangle$.

The Memory Model. LAPOR can be instantiated for any memory model M that satisfies some basic assumptions.

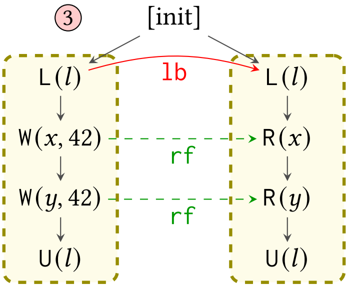


Fig. 9. Illustration of the cs_l and lb relations on an execution of WW+RR-LOCK.

First, M should define a *consistency predicate* on executions, $\mathsf{cons}_\mathsf{M}(.)$ saying which executions are allowed. The permitted behaviours of a given program P under M are described as the set of its *consistent* executions. For the correctness proofs, we require that $\mathsf{cons}_\mathsf{M}(.)$ is *well-formed*, *prefix-closed*, and *lock-extensible* (see §4.2).

Second, we assume that M defines a *happens-before* relation, hb, that contains the po relation and may additionally contain some synchronization edges. For instance, in the case of C11, happens-before contains an edge from a release write w to an acquire read r when r reads from w. Consistency of an execution G must require that hb is a strict partial order (irreflexive and transitive).

***The* wo *Relation*.** Recall from §2 that LAPOR infers the lock order between conflicting critical sections by computing the 'locks-before' relation, lb. As discussed in §2.6, LAPOR is parametric in its notion of write-write conflicts. To model this, we assume the existence of a '*write order*' parameter, wo, describing when two writes on the same location are deemed conflicted. As discussed in §2.6, wo can then be instantiated by either (1) the 'modification order' relation mo, describing a strict *total* order on the writes of each location; or (2) the smaller 'writes-before' relation wb by Lahav and Vafeiadis [2015], described below.

***Notation*.** Given a relation r and a set A, we write $\mathsf{r}^?$, r^+ and r^* for the reflexive, transitive and reflexive-transitive closures of r, respectively. We write $dom(\mathsf{r})$ and $rng(\mathsf{r})$ for the domain and range of r, respectively. We write r^{-1} for the inverse of r; $\mathsf{r}|_A$ for $\mathsf{r} \cap (A \times A)$; $[A]$ for the identity relation on A: $\{(a, a) \mid a \in A\}$; and $[a]$ for $[\{a\}]$. We write r_e to restrict r to its *external* subset, i.e., the r edges between different threads: $\mathsf{r}_\mathsf{e} \triangleq \mathsf{r} \setminus (\mathsf{po} \cup \mathsf{po}^{-1})$. Given relations r_1 and r_2, we write $\mathsf{r}_1 ; \mathsf{r}_2$ for $\{(a, b) \mid \exists c.\ (a, c) \in \mathsf{r}_1 \wedge (c, b) \in \mathsf{r}_2\}$, i.e., their relational composition. When r is a strict partial order, we write $\mathsf{r}|_\mathsf{imm}$ for the *immediate* edges in r, i.e., $\mathsf{r} \setminus \mathsf{r} ; \mathsf{r}$. Given an event set E, we write E_x for $\{e \in E \mid \mathtt{loc}(e) = x\}$. Finally, we write $+\!\!+$ for sequence concatenation.

***The* cs_l *Relation*.** The '*critical section*' relation, cs_l, relates events in the same critical section of lock l. To define it, we introduce an equivalence (reflexive, transitive and symmetric) relation cs_l on events contained in a critical section guarded by lock l. An event a belongs to a critical section guarded by lock l if it is po-after a $\mathtt{L}(l)$ event e and there is no $\mathtt{U}(l)$ event po-between e and a.

Definition 3.3. Given an execution G and a lock l, the *critical section relation on* l, $G.\mathsf{cs}_l$, is defined as follows:

$$G.\mathsf{cs}_l \triangleq \{(a, b) \mid \exists e \in G.\mathtt{L}_l.\ e \xrightarrow{\mathsf{pocs}_l^?} a \wedge e \xrightarrow{\mathsf{pocs}_l^?} b\}$$

with $G.\mathsf{pocs}_l \triangleq G.\mathsf{po} \setminus (G.\mathsf{po}^?; [G.\mathtt{U}_l]; G.\mathsf{po})$.

For example, let us consider the complete execution ③ of the WW+RR-LOCK example of § 2.4 (Fig. 9). As can be seen (highlighted in yellow), the execution has two critical sections: $\mathtt{cs}_l$ relates all the events of the first thread to one another, and all the events of the second thread to one another.

***The* $\mathtt{lb}$ *Relation*.** We move on to the $\mathtt{lb}$ relation, which we incorporate into the underlying model's hb relation. Recall from §2 that an $\mathtt{rf}$ edge between two critical sections in different threads renders the two sections write-read conflicted, and thus induces an $\mathtt{lb}$ edge between the events of the two sections in the $\mathtt{rf}$ direction. Put formally, if a and b are events in a critical section guarded by l, and $(a, b) \in G.\mathtt{rf}_\mathsf{e}$, then for all a', b' such that $(a, a') \in \mathtt{cs}_l$ and $(b, b') \in \mathtt{cs}_l$ (i.e., a' and b' are in the same critical sections as a and b, respectively), we have $(a', b') \in \mathtt{lb}$. It is sufficient to consider the inter-thread $\mathtt{rf}$ edges ($\mathtt{rf}_\mathsf{e}$): same-thread $\mathtt{rf}$ edges either belong to the same critical section or to po-ordered critical sections, in which case they are already ordered by hb.

Similarly, if a read of a critical section a reads from a write that is overwritten by a write in a critical section b of the same lock, then all events in a are $\mathtt{lb}$-before the events in b. Formally, we define the '*reads-before*' relation as: $G.\mathtt{rb} \triangleq G.\mathtt{rf}^{-1}; G.\mathsf{wo}$, and require that $G.\mathtt{cs}_l; G.\mathtt{rb}_\mathsf{e}; G.\mathtt{cs}_l \subseteq G.\mathtt{lb}$. Intuitively, $\mathtt{rb}$ relates each read r to the writes wo-after the write r reads from.

Analogously, in the case of write-write conflicted critical sections, the (parametric) wo relation induces $\mathtt{lb}$ edges in the wo direction. We thus define the $\mathtt{lb}$ relation as the smallest transitive relation that contains the orderings induced by the $\mathtt{rf}_\mathsf{e}$, $\mathtt{rb}_\mathsf{e}$ and wo_e edges between critical sections.

Definition 3.4. Given an execution G, the *locks-before relation*, $G.\mathtt{lb}$, is the smallest transitive relation that satisfies the following for all locations l:

$$G.\mathtt{cs}_l; (G.\mathtt{rf}_\mathsf{e} \cup G.\mathtt{rb}_\mathsf{e} \cup G.\mathsf{wo}_\mathsf{e}); G.\mathtt{cs}_l \subseteq G.\mathtt{lb}$$

We adapt the definition of the underlying memory model, so as to include $\mathtt{lb}$ in the definition of hb. As a result, checking $\mathsf{cons}_\mathsf{M}(.)$ ensures that $\mathtt{lb}$ is irreflexive.

Returning to the execution in Fig. 9, the $\mathtt{rf}$ edges induce an $\mathtt{lb}$ edge from the critical section of the first thread to that of the second thread. Technically, there is an $\mathtt{lb}$ edge from every event of thread 1 to every event of thread 2, but we only show one edge for brevity.

***The* wb *Instantiation of* wo.** As discussed above, the wo parameter may be instantiated as either: (1) the 'modification order', $\mathsf{mo} \triangleq \bigcup_{x \in \mathsf{Loc}} \mathsf{mo}_x$, where each mo_x is a strict total order on the writes of location x; or (2) the 'writes-before' order, $\mathsf{wb} \triangleq \bigcup_{x \in \mathsf{Loc}} \mathsf{wb}_x$, where each wb_x is a strict partial order on the writes of location x, satisfying certain conditions.

We next present the formal definition of wb_x in Def. 3.5, as defined by Lahav and Vafeiadis [2015]. In the original definition by Lahav and Vafeiadis [2015], the $G.\mathsf{hb}$ relation corresponds to the 'happens-before' relation of the release-acquire fragment of the C11 memory model [Batty et al. 2011; Lahav et al. 2017].

Definition 3.5 (The wb *relation).* Given an execution G, its 'happens-before' relation $G.\mathsf{hb}$ and a location x, if $G.\mathsf{hb}$ is transitive and $G.\mathsf{po} \cup G.\mathtt{lb} \subseteq G.\mathsf{hb}$, then the *writes-before relation on* x,$G.\mathsf{wb}_x$, is defined as the smallest transitive relation that satisfies the following inequality:

$$[G.\mathtt{W}_x]; \left((G.\mathtt{rf}^*; G.\mathsf{hb}; (G.\mathtt{rf}^{-1})^?; G.\mathtt{rf}^*) \setminus (G.\mathtt{rf}^{-1})^*\right); [G.\mathtt{W}_x] \subseteq G.\mathsf{wb}_x$$

The *writes-before* relation is defined as: $\mathsf{wb} \triangleq \bigcup_{x \in \mathsf{Loc}} \mathsf{wb}_x$.

***Calculating* hb*,* $\mathtt{lb}$*, and* wb.** Note that the definitions of hb, $\mathtt{lb}$, and wb are mutually recursive. The definition of wb depends on hb, which includes $\mathtt{lb}$, whose definition recursively depends on $\mathsf{wo} \triangleq \mathsf{wb}$. We therefore calculate the three relations together by fixpoint iteration. Starting with $\mathtt{lb} = \mathsf{wb} = \emptyset$ and hb as defined by the underlying memory model, we iteratively add to the $\mathtt{lb}$ and

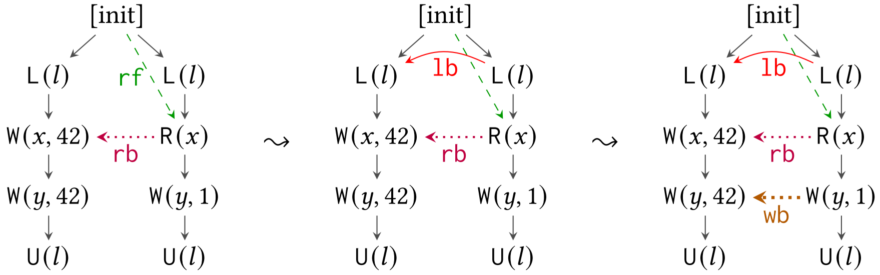


Fig. 10. An example of the recursive calculation of hb, lb, and wb.

wb relations the LHS of the inequalities in Def. 3.4 and Def. 3.5 respectively, as well as any transitive edges, and extend hb to include lb until a fixpoint is reached.

Figure 10 shows an example of the fixpoint calculation. In the first iteration of the fixpoint, wb edges are added from the initializer events to all writes of the graph. (For brevity, Fig. 10 does not display these edges and instead shows the induced rb edge from R(x) to W(x, 42).) In the second iteration, the rb edge induces lb and hb edges from the events of the second thread to the events of the first thread. (Again for brevity, Fig. 10 shows only one such edge.) In the third iteration, a wb edge is created between the W(y, 1) and the W(y, 42), due to the previously created lb/hb edge. In the fourth (and last) iteration, no further edges can be added and so the fixpoint iteration finishes.

4 LAPOR: LOCK-AWARE PARTIAL ORDER REDUCTION

Our model checking algorithm for locks, LAPOR, can be built as an extension of any (D)POR algorithm, as long as it calculates its alternative exploration options at each step and not at the end of each execution. For concreteness, here we present a version of LAPOR built on top of the GenMC model checking algorithm by Kokologiannakis et al. [2019]. Basing LAPOR over GenMC enables our algorithm to be parametric with respect to the memory model and allows us to prove its correctness and optimality (see §4.2).

Configurations. Recall from §2 that given a program P, LAPOR explores the executions of P one at a time, recording the alternative explorations encountered along the way in a *work set*. For instance, when adding a read event r to the current execution G, if r has several reads-from options in G, then one of these options is considered in G, and the remaining options are added to the work set for future exploration. As such, as in GenMC, LAPOR maintains a *configuration* of the form $\langle G, \Gamma \rangle$, where G denotes the current execution, and Γ denotes the work set. A work set Γ in GenMC is a tuple comprising several components in order to track alternative future explorations efficiently. However, as the representation details of work sets are not necessary for understanding LAPOR, here we represent it as an abstract component Γ, and refer the reader to Kokologiannakis et al. [2019] for more details.

The Main* Verify *Procedure. LAPOR begins exploring the executions of a program P by calling Verify(P). This procedure creates an initial configuration comprising the initial graph G_0 (containing initialization events only) and an empty work set Γ. It then generates the executions of P one at a time by calling VisitOne(P, G, Γ) on Line 4, which fully explores one execution extending G, and pushes alternative explorations encountered to the work set Γ. Once VisitOne(P, G, Γ) returns a full execution, remaining executions are generated by exploring the options in Γ (Line 5).

Algorithm 1 Exploration algorithm

```
 1: procedure VERIFY($P$)
 2:     $\langle G, \Gamma \rangle \leftarrow \langle G_0, \varnothing \rangle$
 3:     do
 4:         VISITONE($P, G, \Gamma$)
 5:     while $\langle G, \Gamma \rangle \leftarrow \mathsf{nextExp}(\Gamma)$

 6: procedure VISITONE($P, G, \Gamma$)
 7:     while $\mathsf{cons}_{\mathsf{M}}(G)$ do
 8:         $a \leftarrow \mathsf{nextEvent}_P(G)$
 9:         if $a = \bot$ then
10:             output("Generated execution $G$.")
11:             return
12:         if $a \in \mathtt{error}$ then exit("Erroneous program.")
13:         $G.\mathsf{E} \leftarrow G.\mathsf{E} +\!\!+ [a]$
14:         if $a \in \mathtt{W}$ then
15:             $\mathsf{calcRevisits}(G, \Gamma, a)$
16:         if $a \in \mathtt{R}$ then
17:             $W \leftarrow G.\mathsf{E} \cap \mathtt{W}_{\mathtt{loc}(a)}$
18:             choose some $w_0 \in W$
19:             $G \leftarrow \mathsf{SetRF}(G, w_0, a)$
20:             $\mathsf{push}(\Gamma, \{\mathsf{SetRF}(G, w, a) \mid w \in W \setminus \{w_0\}\})$
```

4.1 The VisitOne Procedure

The VisitOne(P, G, Γ) procedure carries out the crux of the algorithm by exploring one execution at a time. The VisitOne procedure in Algorithm 1 is essentially that of GenMC, but has much more elaborate implementations of the $\mathsf{cons}_{\mathsf{M}}(.)$ and the $\mathsf{nextEvent}_P(.)$ functions (see below).

The VisitOne procedure proceeds as follows. In each iteration of this loop, while the current graph G is M-consistent ($\mathsf{cons}_{\mathsf{M}}(G)$ holds), it is extended with its next event a, given by the $\mathsf{nextEvent}_P(G)$ function described shortly. If $\mathsf{nextEvent}_P(G)$ returns $\bot$, the current execution graph cannot be extended further because the program has either terminated or all the threads are blocked. In this case, VisitOne outputs the execution graph and terminates. If the next event a is an assertion violation, then an error is reported (Line 12), and the algorithm terminates. Otherwise, we add a to the graph by placing it at the end of the sequence of events (Line 13).

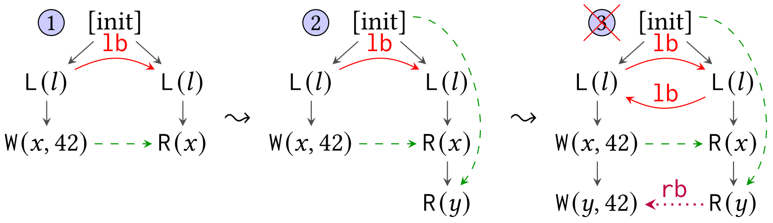
If the new event a is a write, we check whether it may revisit existing reads in the graph; i.e., if existing reads may read from a (see §2.5). This is done by calling $\mathsf{calcRevisits}(G, \Gamma, a)$ on Line 15, which adds all possible revisit options to the work set Γ. For instance, in Fig. 6 of §2.5, upon adding the write event $\mathtt{W}(x, 1)$ to ③, the algorithm detects that $\mathtt{W}(x, 1)$ may revisit the read event $\mathtt{R}(x)$ in the left thread as depicted in ④, and thus ④ is added to the work set. The $\mathsf{calcRevisits}(G, \Gamma, a)$ procedure is that of GenMC and remains unchanged in our LAPOR extension. It is not necessary to understand the details of $\mathsf{calcRevisits}(G, \Gamma, a)$ to understand LAPOR. We have thus omitted the $\mathsf{calcRevisits}(G, \Gamma, a)$ procedure here and refer the reader to [Kokologiannakis et al. 2019].

If a is a read, we must calculate its incoming $\mathtt{rf}$ edge. We first calculate the set of writes W that a could read from (Line 17), choose a write w_0 for the current exploration (Line 18), set a to read from w_0 in G (Line 19), and push the remaining revisit options to Γ (Line 20).

Checking Consistency: The $\mathsf{cons}_{\mathsf{M}}(.)$ ***Function.*** As mentioned in §3, $\mathsf{cons}_{\mathsf{M}}(G)$ includes a check that $\mathtt{hb}$ is irreflexive. Further recall that we have adapted the definition of $\mathtt{hb}$ to include $\mathtt{lb}$, which in turn also depends on $\mathtt{wo}$ (which, in the case of $\mathtt{wo} \triangleq \mathtt{wb}$, recursively depends on $\mathtt{hb}$). Therefore, as part of $\mathsf{cons}_{\mathsf{M}}(G)$, we calculate the three relations together via a fixpoint iteration as explained at the end of §3.

The $\mathsf{nextEvent}_P$ ***Function.*** As described in §2, LAPOR explores each execution of a given program incrementally by adding one event at a time, respecting the program-order (po) in each thread; i.e., the events of each thread are added in po order. To do this, LAPOR uses the $\mathsf{nextEvent}_P$ function of GenMC (Line 8) which returns the next event a to be added to G. More concretely, given a program P and an execution G of P, the $\mathsf{nextEvent}_P(G)$ function in GenMC returns an event a of thread i in G such that: (1) a is is the next event of i in $G.\mathsf{po}$ (i.e., $a.\mathtt{tid} = i$ and $a.\mathsf{sn} = |G.\mathsf{E}_i| + 1$); (2) i is not stuck (e.g., due to a failed **assume** statement); and (3) i has not finished execution. When no such event exists in G (i.e., all threads are stuck or finished), then $\mathsf{nextEvent}_P(G)$ returns $\perp$.

In the presence of locks, defining $\mathsf{nextEvent}_P$ requires additional care. Recall the WW+RR-LOCK program in §2.4 and consider a run of LAPOR where the $lock(l)$ events and the $x := 42$ and $a := x$ events have been added to the execution, with $a := x$ reading from $x := 42$, as shown in ① below.



Note that as discussed in §2, the $\mathtt{rf}$ edge between $x := 42$ and $a := x$ induces an $\mathtt{lb}$ edge between the two critical sections. Given the conditions outlined above, the $\mathsf{nextEvent}_P(G)$ may next choose to add either $y := 42$ or $b := y$. Let us assume that $\mathsf{nextEvent}_P(G)$ chooses $b := y$. As depicted above in ②, since the execution thus far contains no other writes on y, the $b := y$ read has only one reads-from option: the initialization event. However, as shown in ③, reading from the initialization leads to an inconsistent execution: when $y := 42$ is eventually added to the graph, it results in an $\mathtt{rb}$ edge between $b := y$ and $y := 42$, which in turn induces an $\mathtt{lb}$ between the two critical sections, resulting in an $\mathtt{lb}$ cycle. As such, starting from execution ①, we fail to generate the execution in which $a := x$ and $b := y$ both read 42 from the first thread. In other words, the only way to extend ① consistently is for $b := y$ to read from $x := 42$, and thus $\mathsf{nextEvent}_P(G)$ must add $x := 42$ *before* $b := y$. Intuitively, when two critical sections are $\mathtt{lb}$-ordered, $cs_1 \xrightarrow{\mathtt{lb}} cs_2$, the events in cs_1 may affect the reads-from options of those in cs_2, and thus all events of cs_1 must be added before those of cs_2.

More generally, we define $\mathsf{nextEvent}_P(G)$ to prioritize the events of *open critical sections* (those without an *unlock* event), and to add such events in the $\mathtt{lb}$ order. That is, when $\mathsf{nextEvent}_P(G)$=a, as well as the (1)-(3) conditions stipulated by GenMC outlined above, we require that: (4) either G contain no open critical sections, or a be an event in an $\mathtt{lb}$-minimal open critical section:

$$G.\mathsf{OCS}=\emptyset \lor a \in G.\mathsf{OCS} \land \nexists b.\ b \in G.\mathsf{OCS} \land (b, a) \in G.\mathtt{lb}$$

$$\text{where} \quad G.\mathsf{OCS} \triangleq \bigcup_{x \in \mathsf{Loc}} G.\mathsf{OCS}_x \quad \text{with} \quad G.\mathsf{OCS}_x \triangleq \{e \mid [e]; G.\mathtt{cs}_x \neq \emptyset \land [e]; G.\mathtt{cs}_x; [G.\mathsf{U}_x] = \emptyset\}$$

In our LAPOR development, we implement $\mathsf{nextEvent}_P$ to choose an event a in the left-most thread i (i.e., one with the smallest thread identifier) that satisfies conditions (1)-(4).

Note that if the executions of a program P are not well-formed in their acquisition of locks, then $\mathsf{nextEvent}_P$ may get stuck. For instance, consider a variant of the WW+RR-LOCK program in §2.4 without $unlock(l)$ in the left thread. Continuing from graph ① above, after adding the $y := 42$ event, the $\mathsf{nextEvent}_P$ function gets stuck: we cannot add the events of the right thread since the left critical section is open and is $\mathtt{lb}$-before the right one.

To address this, when $\mathsf{nextEvent}_P(G) = \bot$, LAPOR checks that the current execution G is *lock-well-formed*, in that *lock* and *unlock* events are paired. Put formally, let $G.\mathtt{lpo}_x \triangleq (G.\mathsf{po}|_{\mathsf{L}_x \cup \mathsf{U}_x})|_{\mathrm{imm}}$; a (full) execution G is lock-well-formed iff:

$$\forall x, a \in G.\mathsf{E}.\ (a \in G.\mathsf{L}_x \Rightarrow \exists u \in G.\mathsf{U}_x.\ (a, u) \in G.\mathtt{lpo}_x) \wedge (a \in G.\mathsf{U}_x \Rightarrow \exists l \in G.\mathsf{L}_x.\ (l, a) \in G.\mathtt{lpo}_x)$$

When a full execution G is not lock-well-formed, LAPOR reports an error.

4.2 LAPOR: Soundness, Completeness and Optimality

In this section, we establish the correctness of LAPOR by showing that its generated execution graphs satisfy three properties: *soundness*, *completeness* and *optimality*. Given a program P and a memory model m, we say that LAPOR generates execution G for P if running $\textsc{Verify}(P)$ outputs G (at Line 10 of $\textsc{VisitOne}$) among other executions. Soundness ensures that if LAPOR generates G for P, then G is consistent (i.e., $\mathsf{cons}_{\mathsf{m}}(G)$ holds); completeness ensures that if G is a consistent execution of P, then LAPOR generates G for P; and optimality ensures that the executions of P generated by LAPOR are pairwise distinct.

The soundness proof is straightforward: LAPOR checks consistency after each step, dropping inconsistent executions (Line 7 of $\textsc{VisitOne}$); as such, it only outputs consistent executions. The optimality of LAPOR follows immediately from the optimality of its parent algorithm GENMC, as established in [Kokologiannakis et al. 2019]. That is, as LAPOR is implemented by extending GENMC's consistency check and next event selection, and these extensions do not affect the optimality of the algorithm, the optimality of LAPOR follows straightforwardly from that of GENMC.

Establishing completeness is, however, more difficult. Even though GENMC is complete, this does not necessarily imply the completeness of its LAPOR extension. This is because in order to achieve completeness, GENMC requires the underlying memory model to be *well-formed*, *prefix-closed* and *extensible*. While the first two properties (described shortly) are preserved in the presence of locks, extensibility is not. In particular, extensibility requires that execution consistency be preserved under extension with events. That is, for all executions G and all events a, if G is consistent, then after adding a to G the execution remains consistent. As we discussed in §2.5, this is not necessarily the case in the presence of locks: the mere addition of a *write* event may render an execution inconsistent. In other words, the extensibility requirement of GENMC is too strong in the presence of locks. We thus weaken the extensibility requirement and show the LAPOR completeness under a weaker requirement: *lock-extensibility*. That is, if the underlying memory model is well-formed, prefix-closed and lock-extensible, then LAPOR is complete.

Well-formedness ensures that consistent executions do not contain $\mathsf{po} \cup \mathtt{rf}$ cycles: for all G, if G is consistent, then $G.\mathtt{porf} \triangleq (G.\mathsf{po} \cup G.\mathtt{rf})^+$ is irreflexive. As argued by Kokologiannakis et al. [2019], this property precludes problematic executions exhibiting the 'out of thin air' behaviour. Moreover, well-formedness ensures that the execution consistency is independent of the order in which events are added: for all $G = \langle E, rf \rangle$, if G is consistent, then $\langle E', rf \rangle$ is also consistent for any permutation E' of E. This is because the order of events in $G.\mathsf{E}$ constitutes auxiliary instrumentation used by the algorithm, and thus execution consistency must be agnostic to this order.

Prefix-closedness ensures that each execution G of a program can be generated by adding its event in any order that extends $G.\mathtt{porf}$. As noted by Kokologiannakis et al. [2019], all well-known memory models including SC [Lamport 1979], TSO [Owens et al. 2009], PSO [SPARC International Inc. 1994], and RC11 [Lahav et al. 2017], satisfy this property. This allows us to *fix* the order in which events are added via the $\mathsf{nextEvent}_P$ function (described above), and thus generate all program executions systematically. Put formally, the prefix-closedness states that given a consistent execution G and a $\mathtt{porf}$-closed set of events $E \subseteq G.\mathsf{E}$ (i.e., $\mathit{dom}(G.\mathtt{porf}; [E]) \subseteq E$), restricting G to those events in E yields a consistent execution, i.e., $\langle E, G.\mathtt{rf}|_E \rangle$ is consistent.

Recall from our discussion above (see the $\mathsf{nextEvent}_P$ function) that $\mathtt{lb}$-ordered critical sections may affect the reads-from options of one another, thereby affecting consistency. That is, when the event addition order does not respect $\mathtt{lb}$, consistency may be compromised. This is indeed the scenario that led to inconsistency in our 2RW-LOCK example of § 2.5: even though the right critical section in execution ① of Fig. 6 is $\mathtt{lb}$-ordered before the left one, its events are added after those of the left one, leading to inconsistency in ③. Lock-extensibility thus weakens the notion of extensibility to allow for such scenarios; i.e., we require extension to preserve consistency *only if* the construction of the execution thus far respects the $\mathtt{lb}$ order. That is, when extending an execution G with an event a (with $a.\mathsf{sn} = |G.\mathsf{E}_{a.\mathtt{tid}}| + 1$), if G is consistent and either a is not in a critical section, or a is in an $\mathtt{lb}$-maximal critical section, then lock-extensibility requires that the graph obtained by adding a to G be consistent. Note that when a is in a non-$G.\mathtt{lb}$-maximal critical section cs, i.e., there exists cs' such that $cs \xrightarrow{\mathtt{lb}} cs'$, then the cs' events in G were added before a, thereby violating the $\mathtt{lb}$ order; we thus do not require the resulting execution to be consistent. Put formally, lock-extensibility states that for all G, a and G'=$\mathsf{add}(G, a)$, if G is consistent and $\exists l.\ a \in G'.\mathtt{cs}_l \Rightarrow \nexists b.\ (a, b) \in G.\mathtt{lb}$, then either (1) $a \notin G'.\mathsf{R}$ and G' is consistent; or (2) there exists $w \in G'$ such that $G'.\mathtt{rf}[a \mapsto w]$ is consistent.

Theorem 4.1 (Correctness). *The LAPOR algorithm is* sound *and* optimal. *If the underlying memory model is well-formed, prefix-closed and lock-extensible, then LAPOR is complete.*

Proof (sketch). Soundness follows immediately from the VisitOne algorithm, while optimality follows immediately from the optimality of GenMC in Kokologiannakis et al. [2019].

To show that LAPOR is complete, we must show that it generates all executions of a program P under a given memory model M that is well-formed, prefix-closed and lock-extensible. As discussed above and demonstrated in [Kokologiannakis et al. 2019], the well-formedness and prefix-closedness of M ensure that every execution of P can be generated incrementally, by adding one event at a time. We then demonstrate that each execution G of P generated incrementally can also be generated by LAPOR if we permute the order in which its events are added. That is, for each execution G obtained by adding events in the order σ=$e_1, e_2, \cdots, e_n$, there exists some permutation σ' of σ, such that LAPOR adds events in the σ' order and arrives at G. In order to show that such a permutation exists, we may need to remove events and re-add them later (corresponding to the $\mathsf{calcRevisits}(.)$ step). As such, we must ensure that we preserve execution consistency after re-adding each event. This can always be done thanks to the lock-extensibility of M which ensures that LAPOR never gets stuck. More concretely, at each step when adding event a to G_1 and obtaining G_2, either a is not in a critical section in which case consistency of G_2 is ensured by lock-extensibility, or a is in a critical section. In the latter case when a is in a critical section cs, we add a in a way that renders cs a *maximal* critical section in G_2, and thus lock-extensibility ensures that G_2 is consistent. To ensure that cs is deemed a maximal critical section, we proceed as follows. If a is a read event, we pick its reads-from option to be a $\mathtt{wo}$-maximal write in G_1; if a is a write event, we set a to be a $\mathtt{wo}$-maximal event in G_2; and if a is a lock event, we set a to be an $\mathtt{lb}$-maximal event in G_2. Given the definition of $\mathtt{lb}$, this then ensure that the critical section of a, namely cs, is a maximal critical section in G_2.

The detailed completeness proof is given in the Appendix. □

5 EVALUATION

To evaluate the performance of our approach, we have implemented LAPOR as a verification tool for C programs that use C11 atomics and the pthread library for concurrency. More specifically, our LAPOR implementation is an extension of the open-source GenMC model checker [Kokologiannakis et al. 2019], available at https://github.com/MPI-SWS/genmc. We note that since GenMC supports a DPOR algorithm that computes wb for ordering writes on the same memory location, we have instantiated LAPOR based on this DPOR with wb.

In the following, we compare LAPOR against the DPOR algorithm that GenMC employs. We avoid direct comparisons with other SMC tools for two reasons. First, since LAPOR is built over GenMC, the direct comparison between the two tools better demonstrates the pros and cons of the lock-aware approach, while including other tools in the tables would dilute the message. Second, GenMC has already been extensively evaluated against other SMC tools (see Kokologiannakis et al. [2019]) and demonstrated to outperform them in most cases. Although we do not report the measurements, we have confirmed that the same holds for the benchmarks of this paper. In particular, because of GenMC's optimality with respect to a coarse equivalence relation, none of the existing SMC tools can be faster except perhaps by a polynomial factor.

Experimental Setup. We conducted the experiments on a Dell PowerEdge M620 blade system with two Intel Xeon E5-2667 v2 CPU (8 cores @ 3.3 GHz) and 256GB of RAM running a custom Debian-based distribution. We used LLVM 3.8.1 for GenMC. Unless explicitly noted otherwise, all reported times are in seconds. We set the timeout limit to six hours.

5.1 Benchmark Selection

To evaluate the benefit of our approach over the baseline GenMC, we have used a diverse set of benchmarks.

First (§5.2), in order to measure the overhead of calculating the lb relation, we have chosen a set of benchmarks where we expect that LAPOR will have no benefit over GenMC (i.e., the coarser-grained equivalence partitioning of LAPOR will coincide with that of GenMC). For this purpose, we took a standard set of benchmarks from the TACAS competition on Software Verification [SV-COMP 2019] that have been used extensively in the DPOR literature by many different tools [Abdulla et al. 2015, 2016, 2018; Alglave et al. 2013; Huang and Huang 2016]. These benchmarks use locks scarcely and only in a very fine-grained fashion, and so we do not expect any performance improvement by using LAPOR over GenMC.

More specifically, we took the benchmarks from two different concurrency categories: pthread and pthread-atomic. From these, we excluded those that do not contain locks (as they are not interesting for our purposes), as well as those that are not data-deterministic (because data-determinism is a prerequisite for stateless model checking). For the remaining 11 benchmarks, we modelled verifier directives that indicate certain sections should executed atomically (e.g., the __VERIFIER_atomic_begin() and __VERIFIER_atomic_end() primitives) by having our tool acquire/release a designated lock accordingly. We note that these benchmarks are small C programs (50-100 LoC) with the exception of scull, a toy Linux-kernel driver (~455 LoC).

Next (§5.3), in order to evaluate the pros and cons of our approach, we shift our focus to more synthetic benchmarks. We used benchmarks similar to the ones presented in §2 (e.g., N-READS, N-READS-LOCK, WW+RR-LOCK), both with and without lock-induced synchronization, and we evaluate LAPOR in both versions.

These synthetic benchmarks show (1) that our tool can explore *exponentially fewer* executions than GenMC by exploiting independence at the critical-section level, and (2) that the introduction of additional synchronization in a program does *not* have a significant impact in LAPOR. They also

Table 1. Benchmarks from [SV-COMP 2019]. The LB column is the loop bound used.

		Executions		Time (s)	
	LB	*GenMC*	*LAPOR*	*GenMC*	*LAPOR*
fib_bench	5	16 632	**5922**	11.60	**3.83**
indexer	4	**64**	**64**	**0.22**	2.60
lazy	-	**6**	**6**	**0.04**	**0.04**
queue_ok	10	**2**	**2**	**0.04**	0.28
stack-1	5	**252**	**252**	**0.16**	0.61
stateful01-2	-	6	**4**	**0.04**	**0.04**
triangular	5	16 632	**5922**	11.03	**2.03**
gcd	3	**12**	**12**	**0.04**	0.06
read_write_lock	-	**96**	**96**	**0.08**	0.14
scull	-	1749	**922**	0.25	**0.23**
time_var_mutex	-	**2**	**2**	0.04	**0.03**

provide further measurements showing how the calculation of lb impacts the verification time both in benchmarks that have conflicting critical sections, and in benchmarks that do not.

Finally (§5.4), we provide a more realistic workload for evaluating the performance of LAPOR, by running it on a few coarse-grained and fine-grained implementations of concurrent data structures under two different workloads. The data structures we considered are: (1) concurrent sets implemented as a doubly-linked list, (2) concurrent binary search trees (BSTs), and (3) concurrent hashtables. The coarse-grained versions use a global lock to protect the entire data structure, while the fine-grained versions use hand-over-hand locking for the sets and the BSTs, and per-bucket locking for the hashtables. Our first workload includes only searching, and the second includes both searching and updating. For the implementations requiring a linked list, we ported the linked list implementation used in the Linux kernel.

As we will see, LAPOR achieves exponential reductions not only for synthetic benchmarks, but also for these more realistic workloads, and is able to exhaustively verify client programs of such data structures within a few seconds.

5.2 SV-COMP Benchmarks

We start with the standard benchmarks from SV-COMP [2019], whose results are summarized in Table 1. For each benchmark, the table reports both the number of executions explored by each tool and the time taken to generate them. The LB column contains the loop unrolling bound used where applicable.

As shown in Table 1, the performance of LAPOR is comparable to (and sometimes better than) that of GenMC in these benchmarks. In most cases, GenMC performs somewhat better than LAPOR, but there are several cases where LAPOR explores considerably fewer executions than GenMC, and is therefore much faster.

Let us first focus on benchmarks where GenMC is faster than LAPOR, and explain why this is the case (see the indexer and stack-1 benchmarks). In general, in cases where both tools explore the same number of executions, LAPOR has additional overhead compared to a traditional DPOR technique, as expected. This overhead is induced by the calculation of lb, which can be considerable or insignificant depending on the program under test.

As mentioned in §3, lb includes the parametric wo relation that orders writes, which in our implementation is instantiated as wb. However, wb includes hb, and hb in turn includes lb. This means that in order to calculate `lb` (necessary for the consistency checks), we have to perform a fixpoint calculation. This calculation can be relatively expensive (at least $O(n^3)$ in the number of critical sections and/or conflicting write accesses), and is performed repeatedly. Our LAPOR implementation recalculates `lb` from scratch at each step, whereas GenMC checks consistency in a more incremental fashion. This is reflected in the performance of our tool in relevant test cases.

For example, consider the indexer program. This test has 13 threads concurrently modifying a concurrent array, and locks are used to ensure the mutual exclusion of compare-and-swap (CAS) operations. This, however, results in conflicting critical sections (where the lock order does matter) within for-loops that are executed multiple times for each thread. Due to the large number of threads and the number of critical sections in each thread, large executions are created, in which the calculation of `lb` requires a lot of time.

On the other hand, there are several benchmarks where LAPOR is considerably faster than GenMC (see the fib_bench and triangular benchmarks). As shown, LAPOR is able to leverage the independence between different critical sections, resulting in far fewer executions (explorations), which in turn reduces the verification time as well.

The reason behind this is the structure of the benchmarks. In particular, both benchmarks have two concurrent writer threads manipulating shared locations within a critical section, and a third concurrent "checker" thread ensuring that each location satisfies a particular invariant (predicate) at all times. What is crucial, however, is that the two writers write to *different* locations, and the checker reads each location *individually*, rather than reading all locations in one critical section. As such, LAPOR detects that each read of the checker need not be ordered with respect to the writer that does not write to this particular location, and hence explores significantly fewer executions.

In summary, the overhead LAPOR induces is expected, and does not have a considerable impact on the performance of the tool. We note that most of the [SV-COMP 2019] benchmarks are small programs that use locks for the manipulation of shared locations, even though plain atomic accesses would suffice. Nevertheless, despite the fact that there is little to gain in such small test cases where the cost of each execution is quite low, LAPOR outperforms GenMC in a number of cases. As we show in §5.4, in programs with significantly more complex critical sections, the benefits of using an approach such as LAPOR become even more prevalent.

5.3 Synthetic Benchmarks

We continue with a number of synthetic benchmarks with their results summarized in Table 2, for both non-locking (left) and locking (right) versions of the benchmarks.

As in §5.1, the first important observation here is that LAPOR can explore exponentially fewer executions than GenMC. Indeed, inspecting the locking version of these benchmarks, LAPOR is generally significantly faster than GenMC, as it explores exponentially fewer executions. In the only case where both tools explore the same number of executions (because the ordering between the critical sections matters), namely the rw-lock benchmark, LAPOR maintains comparable performance to GenMC, and is only a constant factor slower than GenMC.

The second important observation here is that the introduction of lock-induced synchronization can only reduce the number of executions explored by LAPOR. As shown in the two columns of §5.1, LAPOR explores the same number or fewer executions when going from the non-locking to the locking version of a benchmark. By contrast, when going from the non-locking to the locking version, GenMC explores exponentially more executions for benchmarks such as nreads, nwrites and nww-rr, while exploring fewer executions only for the rw benchmark. This is because in the rw benchmark each thread unconditionally performs a number of operations, and this is why the introduction of lock-induces synchronization in the locking version reduces the number of explored executions. We note that even the mere presence of empty critical sections can lead GenMC to

Table 2. Non-locking (left) and locking (right) versions of synthetic benchmarks

	Executions		Time (s)	
	GenMC	*LAPOR*	*GenMC*	*LAPOR*
nreads(6)	**1**	**1**	**0.04**	**0.04**
nreads(7)	**1**	**1**	**0.04**	**0.04**
nreads(8)	**1**	**1**	**0.04**	**0.04**
nwrites(6)	**1**	**1**	**0.04**	**0.04**
nwrites(7)	**1**	**1**	**0.04**	**0.04**
nwrites(8)	**1**	**1**	**0.04**	**0.04**
nww-rr(6)	**36**	**36**	**0.04**	**0.04**
nww-rr(7)	**49**	**49**	**0.04**	**0.04**
nww-rr(8)	**64**	**64**	**0.04**	**0.04**
rw(6)	**16 807**	**16 807**	**0.69**	0.82
rw(7)	**262 144**	**262 144**	**16.09**	17.78
rw(8)	**4 782 969**	**4 782 969**	**1014.49**	1083.55

	Executions		Time (s)	
	GenMC	*LAPOR*	*GenMC*	*LAPOR*
nreads-lock(6)	720	**1**	0.14	**0.04**
nreads-lock(7)	5040	**1**	1.17	**0.04**
nreads-lock(8)	40 320	**1**	18.95	**0.04**
nwrites-lock(6)	720	**1**	0.14	**0.04**
nwrites-lock(7)	5040	**1**	1.12	**0.04**
nwrites-lock(8)	40 320	**1**	18.08	**0.04**
nww-lock-rr(6)	720	**6**	0.21	**0.04**
nww-lock-rr(7)	5040	**7**	1.40	**0.04**
nww-lock-rr(8)	40 320	**8**	22.32	**0.04**
rw-lock(6)	**720**	**720**	**0.14**	0.37
rw-lock(7)	**5040**	**5040**	**1.27**	3.10
rw-lock(8)	**40 320**	**40 320**	**21.96**	38.00
complex-lock(2)	42	**6**	**0.04**	**0.04**
complex-lock(3)	11 214	**90**	2.81	**0.11**

Table 3. Coarse-grained (left) and fine-grained (right) data structure benchmarks (seekers workload)

	Executions		Time (s)	
	GenMC	*LAPOR*	*GenMC*	*LAPOR*
cset(5)	120	**1**	0.30	**0.11**
cset(6)	720	**1**	1.57	**0.15**
cset(7)	5040	**1**	11.46	**0.18**
cset(8)	40 320	**1**	134.02	**0.25**
cbst(5)	120	**1**	0.12	**0.08**
cbst(6)	720	**1**	0.55	**0.11**
cbst(7)	5040	**1**	4.12	**0.13**
cbst(8)	40 320	**1**	56.82	**0.16**
chtable(5)	113 400	**1**	81.90	**0.05**
chtable(6)	–	**1**	–	**0.07**
chtable(7)	–	**1**	–	**0.09**
chtable(8)	–	**1**	–	**0.11**

	Executions		Time (s)	
	GenMC	*LAPOR*	*GenMC*	*LAPOR*
fset(5)	120	**1**	0.70	**0.13**
fset(6)	720	**1**	3.97	**0.17**
fset(7)	5040	**1**	29.48	**0.22**
fset(8)	40 320	**1**	296.35	**0.29**
fbst(5)	120	**1**	0.16	**0.09**
fbst(6)	720	**1**	0.90	**0.11**
fbst(7)	5040	**1**	6.76	**0.14**
fbst(8)	40 320	**1**	81.34	**0.17**
fhtable(5)	2	**1**	**0.05**	**0.05**
fhtable(6)	4	**1**	**0.05**	**0.05**
fhtable(7)	8	**1**	**0.05**	0.06
fhtable(8)	16	**1**	**0.07**	**0.07**

explore exponentially more executions as the number of threads increases (this is the case for the complex-locks benchmark), while such critical sections do not have any effect in LAPOR.

Finally, it is worth noting that the calculation of lb does not seem to impose a significant overhead on LAPOR. In the non-locking versions of these benchmarks, both tools perform similarly, even in the case of rw, where the number of explored executions is rather large.

5.4 Data Structure Benchmarks

We conclude our evaluation with data structure benchmarks. The results for a reading workload are shown in Table 3; the results for a mixed (reading and writing) workload are shown in Table 4. Entries with a dash "–" denote that the tool did not finish after six hours (the timeout limit).

First, for the reading workload (see Table 3), LAPOR explores exponentially fewer executions than GenMC for both version of these benchmarks with coarse-grained and fined-grained locking schemes. Since there are no concurrent writes on these data structures, LAPOR does not order any of the critical sections, both for the coarse-grained and the fine-grained versions. By contrast, GenMC does keep track of the lock ordering, which has a significant impact even in the fine-grained version. More specifically, even though a hand-over-hand locking scheme is used for the concurrent

Table 4. Coarse-grained (left) and fine-grained (right) data structure benchmarks (mixed workload)

	Executions		Time (s)	
	GenMC	LAPOR	GenMC	LAPOR
cset(5)	120	**4**	**0.20**	0.50
cset(6)	720	**6**	**1.05**	1.65
cset(7)	5040	**6**	7.83	**2.52**
cset(8)	40 320	**18**	96.20	**4.95**
cbst(5)	120	**5**	**0.12**	0.81
cbst(6)	720	**5**	**0.77**	2.45
cbst(7)	5040	**5**	5.78	**3.32**
cbst(8)	40 320	**5**	76.40	**4.36**
chtable(5)	113 400	**2**	103.59	**0.11**
chtable(6)	–	**4**	–	**0.34**
chtable(7)	–	**8**	–	**0.80**
chtable(8)	–	**16**	–	**2.96**

	Executions		Time (s)	
	GenMC	LAPOR	GenMC	LAPOR
fset(5)	352	**4**	0.87	**0.51**
fset(6)	4590	**6**	12.14	**1.94**
fset(7)	38 536	**6**	121.71	**3.10**
fset(8)	340 348	**18**	2274.48	**5.99**
fbst(5)	120	**5**	**0.20**	1.14
fbst(6)	720	**5**	**1.22**	3.57
fbst(7)	5040	**5**	9.00	**4.78**
fbst(8)	40 320	**5**	102.35	**5.47**
fhtable(5)	**2**	**2**	**0.05**	0.10
fhtable(6)	**4**	**4**	**0.05**	0.34
fhtable(7)	**8**	**8**	**0.06**	0.81
fhtable(8)	**16**	**16**	**0.08**	2.95

set and the concurrent BST, which would, for example, reduce lock contention if this program was running on a real CPU, GenMC still needs to serialize the lock acquisitions of the root node in these data structures in order to guarantee full coverage. Thus, GenMC explores the same number of executions for the coarse-grained and the fine-grained version of most of the test cases.

The concurrent hashtable is the only program where GenMC does better when moving from a coarse-grained locking scheme to a fine-grained one. In this benchmark, each thread searches for two items, and collisions occur for 5 or more threads. GenMC is unable to cope with the large state space of the coarse-grained version of the program, timeouts for 6 threads or more, and it is very slow even for 5 threads. In the fine-grained version of the same benchmark GenMC does much better. Nevertheless the number of executions it explores increases exponentially with the number of threads, while LAPOR explores only 1 execution for both versions.

Next, we move on to the mixed workload including both searching and inserting elements (see Table 4). Similarly for this workload, in most cases LAPOR explores exponentially fewer executions and is significantly faster than GenMC. Despite the increased complexity of calculating `lb` (due to concurrent modifications taking place), LAPOR maintains a very good performance and greatly outperforms GenMC, even though it is a bit slower compared to itself on the previous workload. This is expected because the results of the searches in the data structure now depend on the concurrent additions and hence LAPOR explores more than one execution per benchmark.

Finally, we conclude with two observations about two benchmarks under this workload, namely the concurrent set and the concurrent hashtable. For the concurrent set, observe that the number of executions explored increases when going from the coarse-grained to the fine-grained version. This is because the set is implemented as a sorted doubly-linked list, and when an updater reaches the end of the list attempting to insert an element, it must also acquire the lock on the list head, thus contending with both updaters and seekers attempting to traverse the list. For the concurrent hashtable, due to the collisions and concurrent modifications, tracking the lock order is deemed necessary; hence GenMC and LAPOR explore the same number of executions. As expected, LAPOR is slower than GenMC in this case, although LAPOR is still competitive in terms of performance.

6 RELATED WORK

Stateless Model Checking. Following the success of tools such as Verisoft [Godefroid 1997] and CHESS [Musuvathi et al. 2008] paving the way for Stateless Model Checking, the literature of DPOR algorithms grew rapidly [Abdulla et al. 2014; Flanagan and Godefroid 2005; Kokologiannakis et al. 2017]. Initially, DPOR algorithms partitioned the state space based on Mazurkiewicz traces [Mazurkiewicz 1987] under sequential consistency [Abdulla et al. 2014; Flanagan and Godefroid 2005], or based on Shasha-Snir traces [Shasha and Snir 1988][3] under weak memory models [Abdulla et al. 2015, 2016; Kokologiannakis et al. 2017; Norris and Demsky 2013; Zhang et al. 2015]. Subsequently, algorithms that use coarser equivalence partitioning were proposed first for SC [Aronis et al. 2018; Chalupa et al. 2017], and later for Release-Acquire [Abdulla et al. 2018]. Recently, GenMC [Kokologiannakis et al. 2019] proposed a memory model-agnostic DPOR algorithm that exploits such coarse partitioning. However, despite their effectiveness, these algorithms all utilize independence only at the instruction level, and not at the critical section level.

To our knowledge, the only DPOR algorithm that can utilize independence at the critical section level is CDPOR (Constrained Dynamic Partial Order Reduction) by Albert et al. [2018]. More specifically, CDPOR exploits *conditional independence* between critical sections: two critical sections may be considered independent, depending on the current execution state. To check for such independence, CDPOR uses checks based on pre-generated constraints, which may result in (potentially expensive) state-equivalence checks. For example, consider the following program:

$$\begin{array}{l||l||c||l} lock(l); & lock(l); & & lock(l); \\ a_0 := x; & x := v_1; & \cdots & x := v_n; \\ unlock(l) & unlock(l) & & unlock(l) \end{array} \qquad (\text{R+}N\text{-WRITES})$$

If $v_1 = v_2 = \cdots = v_n$, then CDPOR detects that reading from any of the N writes leads to an equivalent state, and hence explores 2 executions (one reading the initial value, and one reading from one of the writes). On the other hand, LAPOR explores $N + 1$ executions (one for each possible value the read can read), as it does not take conditional independence into account. However, if $v_1, v_2, \ldots, v_n$ are pairwise distinct (and non-zero), CDPOR explores $(N + 1)!$ executions (as the N writes all lead to different states), while LAPOR still explores $N + 1$ executions.

In other words, CDPOR introduces a semantic pruning in order to exploit independence at the critical section level, while LAPOR *extends* the underlying DPOR (which is a form of syntactic pruning) to the critical section level. Indeed, the two techniques are complementary and could be combined, potentially yielding even better reductions.

Unfoldings. Rodríguez et al. [2015] propose a combination of unfoldings and DPOR to reduce the number of explored executions under SC only. Although exponential reductions (compared to Mazurkiewicz traces) can be achieved in this case as well, this work is also another form of semantic pruning, and is orthogonal to the underlying DPOR framework.

Maximal Causality Reduction. Finally, Maximal Causality Reduction (MCR) [Huang 2015; Huang and Huang 2016] is another technique for stateless model checking which is different from (D)POR. MCR uses a maximal causal model to partition the state space into equivalence classes, which can lead to a coarser equivalence partitioning than the one obtained by wb.

Intuitively, given one execution of the program, MCR generates a set of constraints that allow a particular read in the execution to read a different value (rather than read from a different place), and the feasibility of these constraints is checked with an SMT solver. These constraints also include Φ_{lock}, which totally orders the critical sections for each lock location. It may well be possible to replace Φ_{lock} with a constraint similar to lb acyclicity that uses MCR's data validity constraints to deduce necessary lock orderings.

[3] Shasha-Snir traces are the natural extension of Mazurkiewicz traces for weak-memory consistency.

Race Detection. The idea of keeping critical sections unordered has also been applied in the context of race detection [Kini et al. 2017; Roemer et al. 2018; Smaragdakis et al. 2012]. Given a trace of an execution of a concurrent program (under SC), these techniques attempt to discern whether there exists a pair of unordered, conflicting events, which constitutes a race. However, instead of using hb for ordering events, critical sections are only ordered when necessary (via similar relations to lb), enabling the detection of more races from one given trace.

While these works are typically sound (i.e., no false positives), they are not complete, as they only guarantee that a race does not exist in the given trace, and in all traces that can be inferred from the given one (by taking all linear extensions of the partial lock ordering). By contrast, our approach is both sound and complete, although verification is in general more expensive.

7 CONCLUSIONS AND FUTURE WORK

In this paper, we presented LAPOR, a new stateless model checking approach for handling programs with locks. We demonstrated how LAPOR reduces the umber of explorations exponentially by recording the lock ordering only when necessary. LAPOR achieves this by identifying read-write conflicted critical sections, and exploiting the underlying (D)POR technique and utilizing its notion of conflicts to detect write-write-conflicted critical sections. In comparison to classical DPOR approaches, LAPOR explores far fewer executions for programs that use locks conservatively, i.e., programs that acquire locks more often than strictly necessary. This, however, means that the the time spent per execution is longer, as the consistency condition checked is more complex.

More interestingly, LAPOR can be used to model check *transactional* programs under the *serializability* consistency model [Papadimitriou 1979]. As with locks, serializability requires that concurrent transactions (critical sections) appear to execute one after the other in a total sequential order. Each serializable transaction is of the form [C], where C denotes a block of code to be executed atomically, tantamount to a critical section delimited by locks. More concretely, a transactional program P under serializability can be converted into an equivalent locking program by replacing each transaction [C] in P, with $lock(l)$; C; $unlock(l)$ instead, where l denotes a global lock acquired by all transactions. As such, LAPOR can model check transactional programs under serializability by treating each transaction as a critical section.

We believe that the idea of not recording the order between lock events can be generalized to other synchronization idioms, such as reader-writer locks and barriers, and can decrease the verification time for other classes of programs. More generally, the same idea could perhaps be applied to programs with reads whose returned value does not affect the thread's subsequent memory accesses. For such read events, it should be possible to avoid recording the incoming rf-edges, and hence drastically reduce the number of execution graphs explored. In the future, we would like to explore these ideas further.

As an additional direction of future work, we would like to investigate combining LAPOR with CDPOR (constrained dynamic partial order reduction) [Albert et al. 2018]. As discussed in §6, the semantic pruning of CDPOR is orthogonal to the syntactic pruning of LAPOR, and the combination of the two techniques may further reduce the number of explorations.

ACKNOWLEDGMENTS

We thank the OOPSLA'19 reviewers for their valuable feedback. The second author was supported in part by a European Research Council (ERC) Consolidator Grant for the project "RustBelt", under the European Union Horizon 2020 Framework Programme (grant agreement number 683289).

REFERENCES

Parosh 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, 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, Mohamed Faouzi Atig, Bengt Jonsson, and Carl Leonardsson. 2016. Stateless Model Checking for POWER. In *CAV 2016 (LNCS)*, Vol. 9780. Springer, Berlin, Heidelberg, 134–156. https://doi.org/10.1007/978-3-319-41540-6_8

Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, and Tuan Phong Ngo. 2018. Optimal Stateless Model Checking Under the Release-acquire Semantics. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 135 (Oct. 2018), 29 pages. 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 (LNCS)*, Vol. 10982. Springer, Berlin, Heidelberg, 392–410. https://doi.org/10.1007/978-3-319-96142-2_24

Jade Alglave, Daniel Kroening, and Michael Tautschnig. 2013. Partial Orders for Efficient Bounded Model Checking of Concurrent Software. In *CAV 2013 (LNCS)*, Vol. 8044. Springer, Berlin, Heidelberg, 141–157. https://doi.org/10.1007/978-3-642-39799-8_9

Stavros Aronis, Bengt Jonsson, Magnus Lång, and Konstantinos Sagonas. 2018. Optimal Dynamic Partial Order Reduction with Observers. In *TACAS 2018 (LNCS)*, Vol. 10806. Springer, Berlin, Heidelberg, 229–248. https://doi.org/10.1007/978-3-319-89963-3_14

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

Marek Chalupa, Krishnendu Chatterjee, Andreas Pavlogiannis, Nishant Sinha, and Kapil Vaidya. 2017. Data-centric Dynamic Partial Order Reduction. *Proc. ACM Program. Lang.* 2, POPL, Article 31 (Dec. 2017), 30 pages. https://doi.org/10.1145/3158119

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

Patrice Godefroid. 1997. Model Checking for Programming Languages using VeriSoft. In *POPL 1997*. ACM, New York, NY, USA, 174–186. https://doi.org/10.1145/263699.263717

Jeff Huang. 2015. Stateless model checking concurrent programs with maximal causality reduction. In *PLDI 2015*. ACM, New York, NY, USA, 165–174. https://doi.org/10.1145/2737924.2737975

Shiyou Huang and Jeff Huang. 2016. Maximal Causality Reduction for TSO and PSO. In *OOPSLA 2016*. ACM, New York, NY, USA, 447–461. https://doi.org/10.1145/2983990.2984025

Dileep Kini, Umang Mathur, and Mahesh Viswanathan. 2017. Dynamic Race Prediction in Linear Time. In *PLDI 2017*. ACM, New York, NY, USA, 157–170. https://doi.org/10.1145/3062341.3062374

Michalis Kokologiannakis, Ori Lahav, Konstantinos Sagonas, and Viktor Vafeiadis. 2017. Effective Stateless Model Checking for C/C++ Concurrency. *Proc. ACM Program. Lang.* 2, POPL, Article 17 (Dec. 2017), 32 pages. https://doi.org/10.1145/3158105

Michalis Kokologiannakis, Azalea Raad, and Viktor Vafeiadis. 2019. Model Checking for Weakly Consistent Libraries. In *PLDI 2019*. ACM, New York, NY, USA, 15. https://doi.org/10.1145/3314221.3314609

Ori Lahav and Viktor Vafeiadis. 2015. Owicki-Gries Reasoning for Weak Memory Models. In *ICALP 2015 (LNCS)*, Vol. 9135. Springer, Berlin, Heidelberg, 311–323. https://doi.org/10.1007/978-3-662-47666-6_25

Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. 2017. Repairing Sequential Consistency in C/C++11. In *PLDI 2017*. ACM, New York, NY, USA, 618–632. https://doi.org/10.1145/3062341.3062352

Leslie Lamport. 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

Antoni Mazurkiewicz. 1987. Trace Theory. In *Petri nets: Applications and relationships to other models of concurrency (LNCS)*, Vol. 255. Springer, Berlin, Heidelberg, 279–324. https://doi.org/10.1007/3-540-17906-2_30

Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gérard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. 2008. Finding and Reproducing Heisenbugs in Concurrent Programs. In *OSDI 2008*. USENIX Association, Berkeley, CA, USA, 267–280.

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

Scott Owens, Susmit Sarkar, and Peter Sewell. 2009. A Better x86 Memory Model: x86-TSO. In *TPHOLs 2009*. Springer, 391–407. https://doi.org/10.1007/978-3-642-03359-9_27

Christos H. Papadimitriou. 1979. The Serializability of Concurrent Database Updates. *J. ACM* 26, 4 (Oct. 1979), 631–653. https://doi.org/10.1145/322154.322158

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

Jake Roemer, Kaan Genç, and Michael D. Bond. 2018. High-coverage, Unbounded Sound Predictive Race Detection. In *PLDI 2018*. ACM, New York, NY, USA, 374–389. https://doi.org/10.1145/3192366.3192385

Dennis Shasha and Marc Snir. 1988. Efficient and Correct Execution of Parallel Programs That Share Memory. *ACM Trans. Program. Lang. Syst.* 10, 2 (April 1988), 282–312. https://doi.org/10.1145/42190.42277

Yannis Smaragdakis, Jacob Evans, Caitlin Sadowski, Jaeheon Yi, and Cormac Flanagan. 2012. Sound Predictive Race Detection in Polynomial Time. In *POPL 2012*. ACM, New York, NY, USA, 387–400. https://doi.org/10.1145/2103656.2103702

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

SV-COMP. 2019. Competition on Software Verification (SV-COMP). https://sv-comp.sosy-lab.org/2019/ [Online; accessed 27-March-2019].

Naling Zhang, Markus Kusano, and Chao Wang. 2015. Dynamic partial order reduction for relaxed memory models. In *PLDI 2015*. ACM, New York, NY, USA, 250–259. https://doi.org/10.1145/2737924.2737956

A PROOFS

A.1 Definitions

A.1.1 Graph Relations.

Definition A.1. Given an execution G and a lock l, the *unlock relation* on l, $G.\mathtt{ul}_l$, is defined as follows:

$$G.\mathtt{ul}_l \triangleq \{(a,b) \mid \mathtt{typ}(a) = \mathsf{L} \land \mathtt{typ}(b) = \mathsf{U} \land (a,b) \in G.\mathtt{cs}_l\}$$

Intuitively, the unlock relation matches lock events to their corresponding unlock events.

In what follows, we write $<_G$ for the strict total order induced by the $G.\mathsf{E}$ sequence. That is, $a <_G b$ iff a appears before b in $G.\mathsf{E}$.

A.1.2 Memory model.

Proposition A.2 (Read-extensibility). *Given execution G such that* $\mathsf{cons}(G)$ *holds, and event $r = \langle i, |G.\mathsf{E}_i| + 1, l\rangle$ with* $\mathtt{typ}(r) = \mathsf{R}$, *there exists $w \in G.\mathsf{W}_{\mathtt{loc}(r)}$ such that* $\mathsf{cons}(\mathsf{SetRF}(\mathsf{add}(G, r), w, r))$.

Proposition A.3 (Write-extensibility). *Given execution G and event w, with* $\mathtt{typ}(w) = \mathtt{W}$ *and rng*$([w]; \mathtt{porf}) = \emptyset$, *if* $\mathsf{cons}(G \setminus \{w\})$ *holds, then so does* $\mathsf{cons}(G)$.

Proposition A.4 (Update-extensibility). *Given execution G such that* $\mathsf{cons}(G)$ *holds, and event u with* $\mathtt{typ}(u) = \mathtt{RMW}$ *and rng*$([u]; G.\mathsf{po}) = \emptyset$, *then the following also hold:*

(a) Given event r such that:
- $\mathtt{typ}(r) = \mathsf{R}$
- $\langle u, r\rangle \in G.\mathtt{rf}$
- *rng*$([r]; G.\mathsf{po}) = \emptyset$

then $\exists w \in G.\mathsf{E} \setminus \{u\}$ *such that* $\mathsf{cons}(\mathsf{SetRF}(G, w, r))$.

(b) Given events u', w such that:
- $\mathtt{typ}(u') = \mathtt{RMW}$ *and* $\mathtt{typ}(w) \in \{\mathtt{W}, \mathtt{RMW}\}$
- $\langle u, u'\rangle, \langle w, u\rangle \in G.\mathtt{rf}$
- *rng*$([u']; G.\mathtt{porf}) = \emptyset$

then $\mathsf{cons}(\mathsf{SetRF}(G \setminus \{u\}, w, u'))$.

Proposition A.5 (Extensibility). *Given execution G such that* $\mathsf{cons}(G)$ *holds, an event a, and G'=*$\mathsf{add}(G, a)$, *then if $\nexists b.\ (a,b) \in G.\mathtt{lb}$, then one of Propositions A.2 to A.4 holds for G and a.*

Definition A.6 (Prefix-closedness). A memory model m is *prefix-closed* iff for all G, $E \subseteq G.\mathsf{E}$, if *dom*$(\mathtt{porf}; [E]) \subseteq E$ and $\mathsf{cons}_{\mathsf{m}}(G)$, then there exists $B \subseteq G.\mathsf{B}$ such that $\mathsf{cons}_{\mathsf{m}}(G|_{E \setminus B})$.

A.1.3 Algorithm.

Definition A.7 (Prefix construction). Given an execution $G_f \supseteq G_0$, an event e is added to G_0 in $\mathtt{porflb}$*-order*, written $G_0 \xrightarrow{e} G_1$, if $e \in G_f \setminus G_0$, and *dom*$(G_f.\mathtt{porflb}; [e]) \subseteq G_0$, and $G_1 \setminus \{e\} = G_0$, and $G_0.\mathtt{OCS} \neq \emptyset \Rightarrow e \in G_1.\mathtt{OCS}$.

A graph G is constructed in $\mathtt{porflb}$*-order* if $\varnothing \rightarrow^* G$, where $\rightarrow^*$ denotes the reflexive-transitive closure of $\rightarrow$ (i.e., G is constructed via a succession of $\mathtt{porflb}$-steps, while opening critical sections one by one).

Definition A.8 (Algorithmic construction). Given an execution G_0, an event e is added to G_0 in *algorithmic order*, written $G_0 \Rightarrow G_1$, if there exist e, T, S, U, T', S', U' such that either:

- VisitOne in Algorithm 1 successfully adds e to G_0 from a configuration $\langle G_0, T, S, U\rangle$ arriving at a configuration $\langle G_1, T', S', U'\rangle$ (in which case we write $G_0 \overset{e}{\Rightarrow} G_1$, or $G_0 \Rightarrow_{nr} G_1$), or

- Verify in Algorithm 1 calls $\mathsf{nextExp}(\Gamma)$ from a configuration $\langle G_0, T, S, U\rangle$ arriving at a configuration $\langle G_1, T', S', U'\rangle$.

A graph G is constructed in *algorithmic order* if $\varnothing \Rightarrow^* G$, where$\Rightarrow^*$ denotes the reflexive-transitive closure of$\Rightarrow$ (i.e., there is a run of Algorithm 1 that produces a configuration $\langle G, -, -, -\rangle$).

Lastly, given an algorithmic construction $G_0 \Rightarrow^* G_n$, a graph G_i is part of this construction, written $G_i \in [G_0 \Rightarrow^* G_n]$, if it is $G_0 \Rightarrow^* G_i \Rightarrow^* G_n$. We write $|G_0 \Rightarrow^* G_j|$ to denote the length (in number of steps) of the construction $G_0 \Rightarrow^* G_n$, and $[G_0 \Rightarrow^* G_n]_i$ to denote the i^{th} step of the construction (it must be $i \leq n$).

Definition A.9 (Proper extension). Given a program P and executions G^{ext} and G, we say that G^{ext} is a *proper extension* of G in P, written $\mathrm{ext}_\mathrm{P}(\mathrm{G}^{ext}, \mathrm{G})$, if the following hold:

- $G^{ext} \in [\![P]\!]$
- $G \subseteq G^{ext}$

A.2 Completeness Proof

Theorem A.10 (Completeness). *For any $G_f \in [\![P]\!]$ such that* $\mathsf{cons}(G_f)$ *it is $\varnothing \Rightarrow^* G_f$.*

Assume: 1. P is a deterministic program with finite traces
2. $G_f \in [\![P]\!]$
3. $\mathsf{cons}(G_f)$

Prove: $\varnothing \Rightarrow^* G_f$

$\langle 1\rangle 1.$ $\varnothing \rightarrow^* G_f$

Proof: Can be shown trivially by induction on the length of G_f.

$\langle 1\rangle 2.$ Q.E.D.

Proof: From Lemma A.11, by taking $G = G_f = G^{ext}$.

□

Lemma A.11 (Prefix extension). *If $\varnothing \rightarrow^* G \Rightarrow^* G^{ext}$ with* $\mathsf{cons}(G)$ *and $G^{ext} \in [\![P]\!]$ for some program P, then $\varnothing \Rightarrow^* G^{ext}$.*

Proof sketch: Proof by induction: we assume that we have a fixed prefix construction G from which the algorithm can reach G^{ext}, and prove that the algorithm can reach the same extension running from the initial configuration.

For the proof of the inductive step, given a G such that $\varnothing \rightarrow^* G \xrightarrow{e} G'$, and a graph G'^{ext} such that $G' \Rightarrow^* G'^{ext}$, we show that it is also $G \Rightarrow^* G'^{ext}$, and then use the inductive hypothesis to prove that $\varnothing \Rightarrow^* G'^{ext}$.

Assume: 1. P is a deterministic program with finite traces
2. $\varnothing \rightarrow^* G$ with $\mathsf{cons}(G)$
3. $G \Rightarrow^* G^{ext}$ with $G^{ext} \in [\![P]\!]$

Prove: $\varnothing \Rightarrow^* G^{ext}$

$\langle 1\rangle 1.$ Case: $G = \varnothing$

Proof: Trivially holds from assumption 3 and case assumption $\langle 1\rangle$.

$\langle 1\rangle 2.$ Case: 1. $\varnothing \rightarrow^* G \xrightarrow{e} G'$
2. *IH*: For any G^{ext} such that $G \Rightarrow^* G^{ext}$, it is $\varnothing \Rightarrow^* G^{ext}$

Proof: From A.12 by taking $G_0 = G$, $G_f = G_\mathrm{e}$, $G = G$ and $G' = G'$ we get that $G \Rightarrow^* G_f$. Then, we get the desired result from the *IH*.

$\langle 1\rangle 3.$ Q.E.D.

Proof: Steps $\langle 1\rangle 1$, $\langle 1\rangle 2$, assumptions 1-3, and mathematical induction.

□

Proposition A.12 (Commutativity). *If $G_0 \Rightarrow^* G \xrightarrow{e} G' \Rightarrow^* G_f$ for some G_0, G_f with* $\mathsf{cons}(G_0)$, $\mathsf{cons}(G_f)$, *and $G_f \in [\![P]\!]$, it is $G_0 \Rightarrow^* G_f$.*

Proof sketch: We show that running the algorithm from G will also lead to G_f, thus proving that $G_0 \Rightarrow^* G_f$. For that, we do proof by induction on the length of the algorithmic construction $G' \Rightarrow^* G_f$.
For the proof of the inductive step, we show that the algorithmic steps taken when starting from G either commute with e (or the critical section in which e belongs), and then use the inductive hypothesis for a production of a smaller length, or that they will result into a graph that is also produced by the algorithm when starting from G'.
[For convenience, we assume that when a write w in a critical section cs_1 revisits a read r in a critical section cs_2, the algorithmic construction$\Rightarrow$ will first close cs_1, even if the critical sections are on different locks.]

Assume: 1. P is a deterministic program with finite traces
2. $G_0 \Rightarrow^* G$
3. $G \xrightarrow{e} G'$
4. $G' \Rightarrow^* G_f$

Prove: $G_0 \Rightarrow^* G_f$

⟨1⟩1. Case: $G' = G_f$

Proof: Algorithm 1 can simply take the step; it is $G \overset{e}{\Rightarrow} G'$. From case hypothesis ⟨1⟩ and assumption ⟨0⟩:4 we get that $G_0 \Rightarrow^* G_f$.

⟨1⟩2. Case: 1. $G' \overset{a}{\Rightarrow} G_2 \Rightarrow^* G_f$
2. *IH*: For any G_3 such that $G_0 \Rightarrow^* G_3 \xrightarrow{e} G'_3 \Rightarrow^* G_f$, it is $G_0 \Rightarrow^* G_f$

⟨2⟩1. It is $G \overset{b_0}{\Longrightarrow} \hat{G}$

Proof: From graph extensibility and case assumption ⟨0⟩:3, which ensures that there is at least one event to add.

⟨2⟩2. Case: $b_0 = e$

Proof: It is $G \overset{e}{\Rightarrow} G'$ from graph extensibility. Thus, from ⟨1⟩:1 we get that $G_0 \Rightarrow^* G \overset{e}{\Rightarrow} G' \overset{a}{\Rightarrow} G_2 \Rightarrow^* G_f$.

⟨2⟩3. Case: $b_0 = a$ (with $a \neq e$)

⟨3⟩1. Case: $\langle e, a\rangle \notin G_2.\mathtt{rf}$ and a performs no revisiting

⟨4⟩1. There exists G'' such that $G \overset{a}{\Rightarrow} G'' \xrightarrow{e} G_2$

Proof: Graph extensibility (which holds from prefix and algorithmic construction) guarantees that a can be added in a consistent manner in G' and, in addition, in the case where $\mathtt{typ}(a) \in \{\mathtt{R}, \mathtt{RMW}\}$, Def. A.6 with assumption ⟨3⟩ guarantees that $\exists w \in G'', G_2$ such that $\langle w, a\rangle \in G''.\mathtt{rf}$ and $\langle w, a\rangle \in G_2.\mathtt{rf}$ (i.e., $G'' = G_2 \setminus \{e\}$). From case assumptions ⟨1⟩:1 and ⟨3⟩ we get that it is $G'' \xrightarrow{e} G_2$, i.e., the two transitions commute: both $G \overset{a}{\Rightarrow} G'' \xrightarrow{e} G_2$ and $G \xrightarrow{e} G' \overset{a}{\Rightarrow} G_2$ hold.

⟨4⟩2. Q.E.D.

Proof: From ⟨4⟩1 and *IH* (assumption ⟨1⟩:2).

⟨3⟩2. Case: $\langle e, a\rangle \in G_2.\mathtt{rf}$

⟨4⟩1. There is a run of Algorithm 1 such that $G \overset{a}{\Rightarrow}_{nr} G'' \Rightarrow_{nr}{}^* G_1 \overset{e}{\Rightarrow} G_2$

Proof: From Prop. A.15, case assumptions ⟨2⟩, ⟨1⟩:1-2, and ⟨3⟩.

⟨4⟩2. Q.E.D.

From ⟨4⟩1, assumption ⟨0⟩:2 and case assumption ⟨1⟩:1.

⟨3⟩3. Case: a revisits some event $r_0 \neq e$ and $\langle e, a\rangle \notin G_2.\mathtt{rf}$

⟨4⟩1. There exists G_2' such that $G \overset{a}{\Rightarrow}_{nr} G_2'$

Proof: From graph extensibility (Prop. A.3). (Intuitively, this is the graph in which Algorithm 1 will revisit all consistent reads.)

⟨4⟩2. $r_0 <_{G_2'} e$

From assumptions ⟨0⟩:3 and ⟨1⟩:1.

⟨4⟩3. Assume: $e \in dom(G_2'.\mathtt{porf}^?; [a])$

Prove: False

Proof: From case assumption ⟨3⟩ it is $\mathtt{typ}(a) = \mathtt{W}$. Then, case assumption ⟨4⟩ implies that it must be $a \in rng([e]; G_2'.\mathtt{porf}^?; G_2'.\mathtt{po})$. However, this in turn means that $rng([e]; G_2'.\mathtt{porf}) \neq \emptyset$, which contradicts assumption ⟨1⟩:1.

⟨4⟩4. $e \notin G_2$

Proof: When a revisit takes place for all e' such that $e' >_{G_2'} r_0$ and $e' \notin dom(G_2'.\mathtt{porf}^?; [a])$ it is $e' \notin G_2$ (Algorithm 1 and ??). Since for e it also is $e >_{G_2'} r_0$ and $e \notin dom(G_2'.\mathtt{porf}^?; [a])$ (steps ⟨4⟩2 and ⟨4⟩3), it is $e \notin G_2$.

⟨4⟩5. Q.E.D.

Proof: From step ⟨4⟩4 we get that it is $G' \overset{a}{\Rightarrow} G_2$. Then, from case assumption ⟨1⟩:1 and *IH* (assumption ⟨1⟩:2), we get the desired proof.

⟨3⟩4. Case: a revisits e

⟨4⟩1. There exists G'' such that $G' \overset{a}{\Rightarrow} G'' \overset{e}{\rightarrow} G_2$

Proof: Note that, since a revisits e, it must be $\mathtt{typ}(a) \in \{\mathtt{W}, \mathtt{RMW}\}$. However, from prefix construction (assumption ⟨0⟩:3) we get that $\mathtt{typ}(a) = \mathtt{W}$. Thus, from Prop. A.3, we get that $G' \overset{a}{\Rightarrow} G''$. Then, from case assumptions ⟨1⟩:1 and ⟨3⟩, we get that the two transitions commute, i.e., both $G \overset{a}{\Rightarrow} G'' \overset{e}{\rightarrow} G_2$ and $G \overset{e}{\rightarrow} G' \overset{a}{\Rightarrow} G_2$ hold.

⟨4⟩2. Q.E.D.

Proof: From step ⟨4⟩1 and *IH* (assumption ⟨1⟩:2).

⟨3⟩5. Q.E.D.

Proof: Cases ⟨3⟩1–⟨3⟩4 are exhaustive.

⟨2⟩4. Case: $b_0 = b$ (with $b \neq e, a$)

⟨3⟩1. It is $\mathtt{typ}(e) = \mathsf{L}$ and there exists l such that $e \in G'.\mathsf{L}_l$

Proof: Because of a fixed, TID-based $\mathsf{nextEvent}_P$ function, and case hypothesis ⟨2⟩. (Otherwise, it would have been $b_0 = e$ or $b_0 = a$.)

⟨3⟩2. Assume: $|G.\mathtt{OCS}| \neq 0$

Prove: False

Proof: Contradicts ⟨3⟩1 since, if $|G.\mathtt{OCS}| \neq 0$, then e would have been added to an open critical section due to prefix construction.

⟨3⟩3. It is $\nexists a' \in G'.\mathtt{cs}_l$ such that $(e, a') \in G'.\mathtt{lb}$ (the critical section opened by e is maximal)

Proof: Trivially from step ⟨3⟩2.

⟨3⟩4. It is $G' \Rightarrow^* G_u' \overset{u}{\Rightarrow} G_u$ with $\langle e, u\rangle \in G_u.\mathtt{ul}$ and $G_u \Rightarrow^+ G_f$

Proof: Because of steps ⟨3⟩1 and ⟨3⟩3, and a fixed, TID-based $\mathsf{nextEvent}_P$ that respects open critical sections (as defined in §4), it is $G' \Rightarrow^* G_u' \overset{u}{\Rightarrow} G_u$. (Intuitively, the algorithm will close the critical section that e opened.) There is at least one event to be added after G_u due to ⟨2⟩4 and lock-well-formedness.

⟨3⟩5. Case: It is $G_u \overset{c}{\Rightarrow} G_c \Rightarrow^* G_f$ with $c \neq b$

⟨4⟩1. There exists $w \triangleq [G' \Rightarrow^* G_u]_i$ for some i that revisits some event $r \in G.\mathsf{E}$

PROOF: From case assumptions ⟨3⟩ and ⟨2⟩, as well as steps ⟨3⟩1 and ⟨3⟩4.

⟨4⟩2. There exists $G_w \in [G' \Rightarrow^* G'_w \overset{w}{\Rightarrow} G_w \Rightarrow^* G_u]$

PROOF: Trivially from ⟨4⟩1 and case assumption ⟨3⟩. (Note that the critical section of *e* closes anyway due to algorithmic construction.)

⟨4⟩3. There exists a run $G \overset{b}{\Rightarrow} \hat{G} \Rightarrow^* G_w$

⟨5⟩1. It is $G \Rightarrow^* G_e$ with $\mathsf{nextEvent}_P(G) = e$ and for any $e' \in G_e \setminus G$ it is e' is $(\mathtt{wo} \cup \mathtt{rb})$-maximal

PROOF: From Prop. A.13.

⟨5⟩2. It is $G_e \Rightarrow^* G''_w$ following the steps of $[G' \Rightarrow^* G'_w]$ in ⟨4⟩2

PROOF: From Lemma A.14 (using ⟨5⟩1).

⟨5⟩3. Q.E.D.

PROOF: From ⟨5⟩2 $G \Rightarrow^* G_e \Rightarrow^* G''_w \overset{w}{\Rightarrow} G_w$. Note that it is $G''_w \overset{w}{\Rightarrow} G_w$ because of ⟨4⟩1, ⟨5⟩1, and the way a graph is restricted when a revisit takes place.

⟨4⟩4. Q.E.D.

PROOF: From ⟨4⟩3, ⟨4⟩2, case hypothesis ⟨3⟩, and ⟨0⟩:2 we get that $G_0 \Rightarrow^* G_f$.

⟨3⟩6. CASE: It is $G_u \overset{b}{\Rightarrow} G_b \Rightarrow^* G_f$

⟨4⟩1. It is $\nexists i$ such that $w \triangleq [G' \Rightarrow^* G_u]_i$ revisits some event $r \in G.\mathsf{E}$

PROOF: From case assumptions ⟨3⟩ and ⟨2⟩, as well as steps ⟨3⟩1 and ⟨3⟩4.

⟨4⟩2. CASE: $\mathtt{typ}(b) \neq \mathsf{L}$

⟨5⟩1. CASE: b revisits $r_0 < e$

PROOF: It is $G \xrightarrow{e} G' \Rightarrow^* G_u \overset{b}{\Rightarrow} G_b$ and $e \notin G_b$, since all e' such that $e' >_{G_u.\mathsf{E}} r_0$ it is $e' \notin G_b$. Thus it also is $G \overset{b}{\Rightarrow} G_b$, and from case assumptions ⟨1⟩:1 and ⟨0⟩:2 we get that $G_0 \Rightarrow^* G_f$.

⟨5⟩2. CASE: b revisits $r \in \mathit{rng}([e]; G_u.\mathtt{pocs}_l)$

⟨6⟩1. It is $G' \Rightarrow^* G'_r \overset{r}{\Rightarrow} G_r \Rightarrow^+ G_u \overset{b}{\Rightarrow} G_b \Rightarrow^* G_f$ where $|G_b.\mathsf{OCS}_l| = 1$

PROOF: Trivially from case hypothesis ⟨5⟩ and ⟨3⟩. (It is $\nexists u \in G_b.\mathsf{U}$ such that $\langle e, u \rangle \in G_b.\mathtt{ul}$.)

⟨6⟩2. It is $G \overset{b}{\Rightarrow} \hat{G} \xrightarrow{e} G_2 \Rightarrow^* G_f$

PROOF: From ⟨6⟩1 we know that the algorithm can follow the steps in $[G' \Rightarrow^* G'_r]$ starting from G_2 (Def. A.6 ensures that the produced graphs will be consistent). Then, it is $G_2 \Rightarrow^* G''_r \overset{r}{\Rightarrow} G_b$, since G_b will contain all events e' for which it is $e' \in \mathit{dom}(G_b.\mathtt{pocs}_l; [r])$, or $e' \in G.\mathsf{E} \cup \{b\}$ (due to the way graphs are restricted during revisits). From ⟨6⟩1 we get that $G_b \Rightarrow^* G_f$.

⟨6⟩3. Q.E.D.

PROOF: From ⟨6⟩2, it is $G_0 \Rightarrow^* G \overset{b}{\Rightarrow} \hat{G} \xrightarrow{e} G_2 \Rightarrow^* G_f$, and from the *IH* (assumption ⟨1⟩:2), we get the desired result.

⟨5⟩3. CASE: $\nexists w \in \mathit{rng}([e]; G_b.\mathtt{pocs}_l)$ such that $\langle w, b \rangle \in G_b.\mathtt{rf}$, and b performs no revisiting

PROOF: From case hypothesis ⟨5⟩ and step ⟨4⟩1, it is $G \overset{b}{\Rightarrow} \hat{G} \xrightarrow{e} G'' \Rightarrow^* G_b$, i.e., b commutes with the whole critical section of e. Thus from the *IH* (assumption ⟨1⟩:2) we get that $G_0 \Rightarrow^* G_f$.

⟨5⟩4. CASE: $\exists w \in \mathit{rng}([e]; G_b.\mathtt{pocs}_l)$ such that $\langle w, b \rangle \in G_b.\mathtt{rf}$

⟨6⟩1. It is $G' \Rightarrow^* G'_w \overset{w}{\Rightarrow} G_w \Rightarrow^* G_u \overset{b}{\Rightarrow} G_b \Rightarrow^* G_f$ with $\langle w, b \rangle \in G_b.\mathtt{rf}$

PROOF: Trivially from case hypotheses ⟨5⟩ and ⟨3⟩.

⟨6⟩2. It is $G \Rightarrow^* G_e$ with nextEvent$_P(G_e) = e$ and for all $e' \in G_e \setminus G$ is is e' is $(\mathtt{wo} \cup \mathtt{rb})$-maximal

Proof: From Prop. A.13.

⟨6⟩3. It is $G_e \Rightarrow^* G''_w$ following the steps of $[G' \Rightarrow^* G'_w]$

Proof: From Lemma A.14, using ⟨6⟩2.

⟨6⟩4. Q.E.D.

Proof: It is $G''_w \overset{w}{\Longrightarrow} G_w$. From steps ⟨6⟩1, ⟨6⟩2, and ⟨6⟩3, case hypothesis ⟨3⟩, and assumption ⟨0⟩:2, we get that $G_0 \Rightarrow^* G_f$.

⟨5⟩5. Q.E.D.

Proof: Cases ⟨5⟩1–⟨5⟩4 are exhaustive.

⟨4⟩3. Case: $\mathtt{typ}(b) = \mathsf{L}$

⟨5⟩1. It is $G \xrightarrow{e} G' \Rightarrow^* G'_u \overset{u}{\Rightarrow} G_u \overset{b}{\Rightarrow} G_b \Rightarrow^* G'_{u_b} \overset{u_b}{\Longrightarrow} G_{u_b} \Rightarrow^* G_f$ with $\langle e, u\rangle \in G_u.\mathtt{ul}$ and $\langle b, u_b\rangle \in G_{u_b}.\mathtt{ul}$

Proof: From ⟨3⟩4, ⟨3⟩6, case hypothesis ⟨5⟩, and lock-well-formedness.

⟨5⟩2. Case: There exists $w \triangleq [G_b \Rightarrow^* G_{u_b}]_i$ for some i that revisits $r_0 < e$ (and $\nexists w \in rng([e]; G_{u_b}.\mathtt{pocs}_l)$ such that $\langle w, r\rangle \in G_{u_b}.\mathtt{rf}$ for some $r \in rng([b]; G_{u_b}.\mathtt{pocs}_l; [w])$)

⟨6⟩1. Let w_0 be the first such w (smallest i) that revisits $r_0 < e$

Proof: Trivially by case hypothesis ⟨5⟩.

⟨6⟩2. It is $G \xrightarrow{e} G' \Rightarrow^* G_u \overset{b}{\Rightarrow} G_b \Rightarrow^* G'_w \overset{w_0}{\Longrightarrow} G_w \Rightarrow^* G_{u_b}$

Proof: From case hypotheses ⟨5⟩, ⟨5⟩, ⟨3⟩ and step ⟨6⟩1.

⟨6⟩3. For all $e' \in rng([e]; G_u.\mathtt{pocs}_l)$ it is $e' \notin G_w$

Proof: In general, it is for any e' such that $e' >_{G'_w} r_0$ it is $e' \notin G_w$. In addition, because $\langle e, b\rangle \notin G'_w.\mathtt{porf}$ (due to the case hypothesis ⟨5⟩), we get the desired result.

⟨6⟩4. For any $r \in dom(G_w.\mathtt{pocs}_l; [r])$ with $\mathtt{typ}(r) = \mathsf{R}$, there is $w' \in G.\mathsf{E} \cup dom(G_u.\mathtt{porf}; [r])$ such that $\langle w', r\rangle \in G_u.\mathtt{rf}$

Proof: From steps ⟨6⟩2, ⟨6⟩3, and case hypothesis ⟨5⟩.

⟨6⟩5. Q.E.D.

From case hypothesis ⟨5⟩, as well as steps ⟨6⟩2, ⟨6⟩3, and ⟨6⟩4 we get that $G \overset{b}{\Rightarrow} G'_b \Rightarrow^* G''_w \overset{w_0}{\Longrightarrow} G_w$. From case hypothesis ⟨1⟩:1 and ⟨0⟩:2 we get that $G_0 \Rightarrow^* G_f$.

⟨5⟩3. Case: There exists $w \triangleq [G_b \Rightarrow^* G_{u_b}]_i$ for some i that revisits $r \in rng([e]; G_u.\mathtt{pocs}_l)$

⟨6⟩1. Let r_0 be the $\mathtt{po}$-earlier read that gets revisited, and w_0 the last such w (greatest i) that revisits r_0

Proof: Trivially from case hypothesis ⟨5⟩.

⟨6⟩2. It is $G' \Rightarrow^* G'_r \overset{r_0}{\Longrightarrow} G_r \Rightarrow^* G_u \overset{b}{\Rightarrow} G_b \Rightarrow^* G'_w \overset{w_0}{\Longrightarrow} G_w \Rightarrow^* G_{u_b} \Rightarrow^* G_f$ where $|G_w.\mathsf{OCS}| = 2$ and $|G_{u_b}.\mathsf{OCS}| = 1$

Proof: Trivially from step ⟨6⟩1 and case hypotheses ⟨5⟩ and ⟨3⟩.

⟨6⟩3. It is $G \overset{b}{\Rightarrow} \hat{G} \Rightarrow^* G'_{w_1} \overset{w_0}{\Longrightarrow} G_{w_1} \Rightarrow^* G'_{u_b} \xrightarrow{e} G_2 \Rightarrow^* G_f$

Proof: Since b opens a maximal critical section, it is $G \overset{b}{\Rightarrow} \hat{G} \Rightarrow^* G'_{w_1} \overset{w_0}{\Longrightarrow}_{nr} G_{w_1} \Rightarrow^* G'_{u_b}$ by following the steps of $G_b \Rightarrow^* G'_w \overset{w_0}{\Longrightarrow} G_w \Rightarrow^* G_{u_b}$, with no revisiting of r_0 taking place (consistency is guaranteed by the maximality of the critical section). From steps ⟨6⟩1, ⟨6⟩2 we know that the algorithm can follow the steps in $[G' \Rightarrow^* G'_r]$ starting from G_2 (Def. A.6 on G_{u_b} ensures that the produced graphs will be consistent). In addition, it is $G_2 \Rightarrow^* G''_r \overset{r}{\Rightarrow} G_{u_b}$, since G_{u_b} will contain all events $e' \in rng([b]; G_{u_b}.\mathtt{pocs}_l)$, all events $e' \in dom(G_{u_b}.\mathtt{pocs}_l; [r_0])$, as well as all the events of $G.\mathsf{E}$ (because case hypothesis $\langle 5\rangle$ ensures that $\langle e, b\rangle \notin G'_w.\mathtt{porf}$, and for any $e' \in rng([e]; G_{w'}.\mathtt{pocs}_l)$ it is $e' >_{G_w} e$). From $\langle 6\rangle 2$ we get that $G_{u_b} \Rightarrow^* G_f$.

$\langle 6\rangle 4$. Q.E.D.

Proof: From $\langle 6\rangle 3$ and from the *IH* (assumption $\langle 1\rangle$:2), we get the desired result.

$\langle 5\rangle 4$. Case: It is $\nexists w \in rng([e]; G_u.\mathtt{pocs}_l)$ that is read by some $r \triangleq [G_b \Rightarrow^* G_{u_b}]_i$ for some i, and $\nexists w \triangleq [G_b \Rightarrow^* G_{u_b}]_j$ for some j that performs revisiting

Proof: From case hypothesis, we get that the two critical sections fully commute: it is $G \overset{b}{\Rightarrow} G'' \Rightarrow^* G''_{u_b} \xrightarrow{e} G_e \Rightarrow^* G_{u_b}$ and $G \xrightarrow{e} G' \Rightarrow^* G_{u_e} \overset{b}{\Rightarrow} G_b \Rightarrow^* G_{u_b}$. Since we know that G_{u_b} can reach G_f (step $\langle 5\rangle 1$) and from the *IH* (assumption $\langle 1\rangle$:2) we get that $G_0 \Rightarrow^* G_f$.

$\langle 5\rangle 5$. Case: There is $w \in rng([e]; G_u.\mathtt{pocs}_l)$ and $r \triangleq [G_b \Rightarrow^* G_{u_b}]_i$ for some i such that r reads from w

$\langle 6\rangle 1$. Let r_0 be the first such r (smallest i) and $w_0 \in rng([e]; G_u.\mathtt{pocs}_l)$ the write where r_0 reads from

Proof: Trivially from case hypothesis $\langle 5\rangle$.

$\langle 6\rangle 2$. It is $G \xrightarrow{e} G' \Rightarrow^* G'_w \overset{w_0}{\Longrightarrow} G_w \Rightarrow^* G_u \overset{b}{\Rightarrow} G_b \Rightarrow^* G'_r \overset{r}{\Rightarrow} G_r \Rightarrow^* G_{u_b} \Rightarrow^* G_f$ with $\langle w, b\rangle \in G_b.\mathtt{rf}$

Proof: Trivially from $\langle 5\rangle 1$, $\langle 6\rangle 1$, and case hypotheses $\langle 5\rangle$ and $\langle 3\rangle$.

$\langle 6\rangle 3$. It is $G \Rightarrow^* G_e$ with $\mathsf{nextEvent}_P(G_e) = e$ and for all $e' \in G_e \setminus G$ is is e' is $(\mathtt{wo} \cup \mathtt{rb})$-maximal

Proof: From Prop. A.13.

$\langle 6\rangle 4$. It is $G_e \Rightarrow^* G'_{w_1}$ following the steps of $[G' \Rightarrow^* G'_w]$

Proof: From Lemma A.14, using $\langle 6\rangle 3$.

$\langle 6\rangle 5$. Q.E.D.

Proof: It is $G'_{w_1} \overset{w}{\Longrightarrow} G_{w_1}$. Then, due to the created $\mathtt{lb}$ ordering (because of the revisit), and due to the $\mathsf{nextEvent}_P$ function, the algorithm will follow the steps of $[G_r \Rightarrow^* G_{u_b}]$, leading to the graph G_r. Finally, from steps $\langle 6\rangle 2$, $\langle 6\rangle 3$, and $\langle 6\rangle 4$ we get that $G_0 \Rightarrow^* G_f$.

$\langle 5\rangle 6$. Q.E.D.

Proof: Cases $\langle 5\rangle 2$–$\langle 5\rangle 5$ are exhaustive.

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

Proof: Cases $\langle 4\rangle 2$ and $\langle 4\rangle 3$ are exhaustive.

$\langle 3\rangle 7$. Q.E.D.

Proof: Cases $\langle 3\rangle 5$ and $\langle 3\rangle 6$ are exhaustive.

$\langle 2\rangle 5$. Q.E.D.

Proof: From $\langle 2\rangle 1$ and case hypothesis $\langle 1\rangle$:1 it is easy to see that cases $\langle 2\rangle 2$–$\langle 2\rangle 4$ are exhaustive.

$\langle 1\rangle 3$. Q.E.D.

Proof: Steps $\langle 1\rangle 1$, $\langle 1\rangle 2$, and mathematical induction.

□

Proposition A.13 (Maximal extension). *If $G \xrightarrow{e} G' \Rightarrow^* G'_u \overset{u}{\Rightarrow} G_u \overset{b}{\Rightarrow} G_b$ with $\mathsf{cons}(G)$, and $|G.\mathsf{OCS}| = 0$, and $\langle e, u\rangle \in G_u.\mathtt{ul}$, and $\nexists w \in [G' \Rightarrow^+ G_u]$ such that w revisits $r \in G.\mathsf{E}$, and $\mathsf{nextEvent}_P(G) = b$, then there is a run of Algorithm 1 from G such that $G \overset{b}{\Rightarrow} G'_b \Rightarrow^* G_1$ with $\mathsf{nextEvent}_P(G_1) = e$, and for any $e' \in G_1.\mathsf{E} \setminus G.\mathsf{E}$ it is e' is maximal in $\mathtt{wo} \cup \mathtt{rb}$.*

Assume: 1. $\mathsf{cons}(G)$
2. $|G.\mathsf{OCS}| = 0$

3. $G \xrightarrow{e} G' \Rightarrow^{+} G_u \overset{b}{\Rightarrow} G_b$ with $\mathtt{typ}(e) = \mathsf{L}$
4. $\nexists w \in [G' \Rightarrow^{+} G_u]$ such that w revisits $r \in G.\mathsf{E}$
5. $\mathsf{nextEvent}_P(G) = b$

PROVE: It is $G \overset{b}{\Rightarrow} G'_b \Rightarrow^{*} G_1$ and the following hold for G_1
1. $\mathsf{nextEvent}_P(G_1) = e$
2. For any $e' \in G_1.\mathsf{E} \setminus G.\mathsf{E}$ it is e' is maximal in $\mathtt{wo} \cup \mathtt{rb}$

⟨1⟩1. ASSUME: Let G_2 be a graph such that the following hold:
1. It is $G \overset{b}{\Rightarrow}\Rightarrow^{*} G_2$
2. $e \notin G_2$
3. For any $e' \in G_2.\mathsf{E} \setminus G.\mathsf{E}$ it is e' is $\mathtt{wo} \cup \mathtt{rb}$-maximal

PROVE: It is $\mathsf{nextEvent}_P(G_2) = e$ or $G_2 \Rightarrow G'_2$ and Items 1–3 hold for G'_2.

⟨2⟩1. CASE: $\mathsf{nextEvent}_P(G_2) = e$

PROOF: We trivially get the desired result from the case hypothesis.

⟨2⟩2. CASE: $\mathsf{nextEvent}_P(G_2) = a$ (with $a \neq e$)

⟨3⟩1. CASE: $\mathtt{typ}(a) = \mathtt{R}$

PROOF: From graph extensibility we know that there exists w such that $\langle w, a\rangle \in G'_2.\mathtt{rf}$, with $G_2 \overset{a}{\Rightarrow} G'_2$ and w being $\mathtt{wo}$-maximal in G_2, G'_2 (a read can always read from the $\mathtt{wo}$-maximal write). Thus, no $\mathtt{rb}$ edges are created by the addition of a, so Items 1–3 hold for G'_2 as well, since $G'_2 \setminus \{e\} = G_2$, and Items 1–3 hold for G_2.

⟨3⟩2. CASE: $\mathtt{typ}(a) = \mathtt{W}$

PROOF: From graph extensibility we know that it is $G_2 \overset{a}{\Rightarrow} G''_2$ with a being $\mathtt{wo}$-maximal in G''_2. In general, it can be the case that $\langle r, a\rangle \in G''_2.\mathtt{rb}$ for some $r \in G_2.\mathsf{E} \setminus G.\mathsf{E}$. However, we also know that there exists G'_2 such that $G_2 \overset{a}{\Rightarrow} G'_2$ with a revisiting the minimal event (in $<_{G_2}$) $r' \in G_2.\mathsf{E} \setminus G.\mathsf{E}$ such that $\mathtt{loc}(r') = \mathtt{loc}(a)$. (It is $\mathsf{cons}(G'_2)$ because from extensibility we get that r can read from the $\mathtt{wo}$-maximal write.) In G'_2, r' is the only read with $\mathtt{loc}(r') = \mathtt{loc}(a)$ such that $r' \in G'_2.\mathsf{E} \setminus G.\mathsf{E}$, and it is $\mathtt{wo} \cup \mathtt{rb}$-maximal (by construction). However, for any $e' \in G'_2.\mathsf{E} \setminus G.\mathsf{E}$ such that $e' \neq r', a$, then e' is also $\mathtt{wo} \cup \mathtt{rb}$-maximal, since $e' \in G_2.\mathsf{E} \setminus G.\mathsf{E}$ too. Thus, Items 1–3 trivially hold for G'_2.

⟨3⟩3. CASE: $\mathtt{typ}(a) \in \{\mathsf{L}, \mathsf{U}\}$

PROOF: From graph extensibility, and since a will not contribute to $\mathtt{wo}$ or $\mathtt{rb}$, we get that $G_2 \overset{a}{\Rightarrow} G'_2$, and Items 1–3 hold for G'_2, since $G'_2 \setminus \{e\} = G_2$, and Items 1–3 hold for G_2.

⟨3⟩4. Q.E.D.

PROOF: Cases ⟨3⟩1–⟨3⟩3 are exhaustive.

⟨2⟩3. Q.E.D.

PROOF: Cases ⟨2⟩1 and ⟨2⟩2 are exhaustive.

⟨1⟩2. Q.E.D.

PROOF: From assumptions ⟨0⟩:1–⟨0⟩:5 we know that Items 1–3 hold for G. Then, because graphs are finite (and based on deterministic programs), we get from ⟨1⟩1 that it is $G \overset{b}{\Rightarrow} G'_b \Rightarrow^{*} G_1$ with $\mathsf{nextEvent}_P(G_1) = e$.

LEMMA A.14 (CS ADDITION). *If $G \xrightarrow{e} G' \Rightarrow^{*} G'_u \overset{u}{\Rightarrow} G_u$ with $\langle e, u\rangle \in G_u.\mathtt{ul}$, and $G_{tgt} \in [G' \Rightarrow^{+} G_u]$, and $G_e \supseteq G$ with $\mathsf{nextEvent}_P(G_e) = e$ and $\forall e' \in G_e.\mathsf{E} \setminus G.\mathsf{E}$ it is e' is $\mathtt{wo} \cup \mathtt{rb}$-maximal, then Algorithm 1 can take the exact same steps in $[G' \Rightarrow^{*} G_{tgt}]$ starting from G_e.*

ASSUME: 1. $\mathsf{cons}(G)$
2. $|G.\mathsf{OCS}| = 0$

3. $G \xrightarrow{e} G' \Rightarrow^* G'_u \overset{u}{\Rightarrow} G_u$ with $\langle e, u\rangle \in G_u.\mathtt{ul}$ and $|G' \Rightarrow^+ G_u| = n$
4. $G_{\mathrm{tgt}} \in [G' \Rightarrow^+ G_u]$ with $|G' \Rightarrow^* G_{\mathrm{tgt}}| = k$ $(k \le n)$
5. $G_e \supseteq G$ for which $\mathsf{nextEvent}_P(G_e) = e$ and $\forall e' \in G_e.\mathsf{E} \setminus G.\mathsf{E}$ it is e' is $\mathtt{wo} \cup \mathtt{rb}$-maximal

Prove: It is $G_e \overset{e}{\Rightarrow} G'_e \Rightarrow^* G'_{\mathrm{tgt}}$ and for all G_i such that $G'_e \Rightarrow^* G_i \Rightarrow^* G'_{\mathrm{tgt}}$ with $|G'_e \Rightarrow^* G_i| = i \le k$ then it is $[G'_e \Rightarrow^* G'_{\mathrm{tgt}}]_i = [G' \Rightarrow^* G_{\mathrm{tgt}}]_i$

⟨1⟩1. Assume: Let G_i be a graph such that the following hold:
1. It is $G_e \overset{e}{\Rightarrow} G'_e \Rightarrow^* G_i$ with $|G'_e \Rightarrow^* G_i| = i \le k$
2. For all $j \le i$ it is $[G'_e \Rightarrow^* G_i]_j = [G' \Rightarrow^* G_{\mathrm{tgt}}]_j$
3. For any $e' \in G_e.\mathsf{E} \setminus G.\mathsf{E}$ it is e' is $\mathtt{wo} \cup \mathtt{rb}$-maximal

Prove: It is $G_i = G'_{\mathrm{tgt}}$ or $G_i \Rightarrow G_{i+1}$ and Items 1–3 hold for G_{i+1}.

⟨2⟩1. Case: $G_i = G'_{\mathrm{tgt}}$

Proof: We trivially get the desired result from the case hypothesis.

⟨2⟩2. Case: $G_i \neq G'_{\mathrm{tgt}}$

Proof: It cannot be $i = k$ because of the case hypothesis and assumptions ⟨1⟩:1–⟨1⟩:2. We will show that $G_i \overset{a}{\Rightarrow} G_{i+1}$ with $a = [G' \Rightarrow^* G_{\mathrm{tgt}}]_{i+1}$. The following cases are exhaustive:

⟨3⟩1. Case: $\mathtt{typ}(a) = \mathtt{R}$

Proof: Graph extensibility and assumption ⟨1⟩:3 we get that it is consistent to add a and that it also is $a = [G' \Rightarrow^* G_{\mathrm{tgt}}]_{i+1}$. Items 1–3 trivially hold for G_{i+1} as well.

⟨3⟩2. Case: $\mathtt{typ}(a) = \mathtt{W}$

Proof: When adding w the algorithm can revisit the minimal event (in $<_{G_i}$, if such exists) $r \in G_i.\mathsf{E} \setminus G.\mathsf{E}$ such that $\mathtt{loc}(r') = \mathtt{loc}(a)$. Revisiting r' is consistent because of assumption ⟨1⟩:3. It trivially holds that $a = [G' \Rightarrow^* G_{\mathrm{tgt}}]_{i+1}$ (since $\mathtt{typ}(a) = \mathtt{W}$), and Items 1–3 hold for G_{i+1} as well.

⟨3⟩3. Case: $\mathtt{typ}(a) \in \{\mathtt{L}, \mathtt{U}\}$

Proof: Graph extensibility guarantees that the addition of a is consistent (lock and unlock events are treated as dummy events).

⟨2⟩3. Q.E.D.

Proof: Cases ⟨2⟩1 and ⟨2⟩2 are exhaustive.

⟨1⟩2. Q.E.D.

Proof: From assumptions ⟨0⟩:1–⟨0⟩:5 we know that Items 1–3 hold for G_e and that $\mathsf{nextEvent}_P(G_e) = e$. Then, because graphs are finite (and based on deterministic programs), from ⟨1⟩1 we get the desired result.

Proposition A.15 (Revisitable extension). *If $\varnothing \rightarrow^* G \xrightarrow{e} G' \overset{a}{\Rightarrow} G_2 \Rightarrow^* G_f$, with $a <_G e$ and $\langle e, a\rangle \in G_2.\mathtt{rf}$, then there is a run of Algorithm 1 from G such that $G \overset{a}{\Rightarrow} G'' \Rightarrow_{nr}{}^* G_1 \overset{e}{\Rightarrow} G_2$.*

Proof sketch: We basically want to prove that when event e is added by the algorithm starting from G, event a will be considered as an option to revisit. Thus, we perform a case analysis on the type of e and show that there is a run where Algorithm 1 will not discard a from the set of revisitable options.

Assume:
1. P is a deterministic program with finite traces
2. $\varnothing \rightarrow^* G \xrightarrow{e} G' \overset{a}{\Rightarrow} G_2 \Rightarrow^* G_f$
3. $a <_G e$
4. $\langle e, a\rangle \in G_2.\mathtt{rf}$

Prove: There exists a run of Algorithm 1 from G such that $G \overset{a}{\Rightarrow}_{nr} G'' \Rightarrow_{nr}{}^* G_1 \overset{e}{\Rightarrow} G_2$

⟨1⟩1. It is $\mathsf{nextEvent}_P(G) = a$

Proof: From assumptions $\langle 0\rangle$:2 and $\langle 0\rangle$:3.

$\langle 1\rangle 2$. There is at least one run $G \overset{a}{\Rightarrow}_{nr} G'' \Rightarrow_{nr}{}^{*} G_1$ with $\mathsf{nextEvent}_P(G'_1) = e$

Proof: Graph extensibility guarantees that Algorithm 1 will be able to add all events a' with $a' <_G e$, resulting to a graph G_0, with no revisits performed. Since Algorithm 1 performs no revisiting, and all events e' such that $e' \in \mathit{dom}(G_2.\mathtt{porf}; [e])$ are already in G (prefix construction; assumption $\langle 0\rangle$:2), it will be $\mathsf{nextEvent}_P(G_0) = e$.

$\langle 1\rangle 3$. Case: $\mathtt{typ}(e) = \mathtt{W}$

$\langle 2\rangle 1$. Choose graphs G'_1 and G_p such that

1. $G \overset{a}{\Rightarrow}_{nr} G_1 \Rightarrow_{nr}{}^{*} G'_1$
2. $G'_1 \overset{e}{\Rightarrow}_{nr} G_p$

Proof: Step $\langle 1\rangle 2$ and graph extensibility (Prop. A.3) guarantee that we can select appropriate G'_1 and G_p.

$\langle 2\rangle 2$. It is $a \in T$ when G'_1 is reached in the production $G \overset{a}{\Rightarrow}_{nr} G_1 \Rightarrow_{nr}{}^{*} G'_1$

Proof: From step $\langle 2\rangle 1$ we know that no revisiting is performed, and only if an event gets revisited it is removed from T.

$\langle 2\rangle 3$. It is $\mathsf{cons}(G_2)$

Proof: By assumption $\langle 0\rangle$:2.

$\langle 2\rangle 4$. Assume: $\langle a, e\rangle \in G_p.\mathtt{porf}$

Prove: False

$\langle 3\rangle 1$. It is $e \in \mathit{rng}([a]; G_p.\mathtt{porf}^?; G_p.\mathtt{po})$

Proof: This implies that revisiting is performed, which contradicts case assumption $\langle 2\rangle$:1.

$\langle 3\rangle 2$. It is $e \in \mathit{rng}([a]; G_p.\mathtt{porf}^?; G_p.\mathtt{rf})$

This implies that $e \in \mathit{rng}([a]; G_p.\mathtt{rf})$, which contradicts case assumption $\langle 1\rangle$.

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

Proof: From assumption $\langle 2\rangle$ we get that cases $\langle 3\rangle 1$ and $\langle 3\rangle 2$ are exhaustive.

$\langle 2\rangle 5$. Q.E.D.

It will be $G'_1 \overset{e}{\Rightarrow} G_2$, because a will be considered as an event to revisit by Algorithm 1, and will not be discarded (steps $\langle 2\rangle 2$, $\langle 2\rangle 3$, $\langle 2\rangle 4$).

$\langle 1\rangle 4$. Case: $\mathtt{typ}(e) = \mathtt{RMW}$

$\langle 2\rangle 1$. Let $w \in G.\mathsf{E}$ such that $\langle w, e\rangle \in G'.\mathtt{rf}$

Proof: Such a w exists in G due to the prefix construction (assumption $\langle 0\rangle$:2).

$\langle 2\rangle 2$. There exists $m \in \mathbb{N}$ and G_0, G_3, G_4 such that

1. $G_0 \Rightarrow_{nr}{}^{m} G_3$
2. $\mathtt{tid}(\mathsf{nextEvent}_P(G_3)) = \mathtt{tid}(e)$
3. One of the following holds
   a. $G_0 = G'$ and $\nexists r \in G_3.\mathtt{rf}$ such that $\langle e, r\rangle \in G_3.\mathtt{rf}$
   b. $G_0 = G' \setminus \{e\}$ and $e \notin G_3.\mathsf{E}$ and $\exists u' \in G_3.\mathsf{E}$ such that $\langle w, u'\rangle \in G_3.\mathtt{rf}$

Proof: From assumption $\langle 0\rangle$:2, case assumption $\langle 1\rangle$, step $\langle 2\rangle 1$, and by choosing an appropriate n (which is always possible due to program finiteness), Prop. A.16 yields the required result.

$\langle 2\rangle 3$. Choose G'_1 such that $G'_1 = G_3 \setminus \{e\}$, $G \Rightarrow_{nr}{}^{m} G'_1$, and one of the following holds:

1. $\nexists r \in G'_1.\mathsf{E}$ such that $\langle e, r\rangle \in G'_1.\mathtt{rf}$
2. $\exists u' \in G'_1.\mathsf{E}$ such that $\langle w, u'\rangle \in G'_1.\mathtt{rf}$

$\langle 3\rangle 1$. Choose G'_1 such that $G'_1 = G_3 \setminus \{e\}$

Proof: From Def. A.6 and $\langle 2\rangle 2.3$.

$\langle 3\rangle 2$. Case: $G_0 = G'$ and $\nexists r \in G_3.\mathsf{E}$ such that $\langle e, r\rangle \in G_3.\mathtt{rf}$

$\langle 4\rangle 1$. It is $G_0 \setminus \{e\} \Rightarrow_{nr}{}^{m} G'_1$ and $\nexists r \in G'_1.\mathsf{E}$ such that $\langle e, r\rangle \in G'_1.\mathtt{rf}$

Proof: From step $\langle 2\rangle 2.1$ and case assumption $\langle 3\rangle$ we know that we can use Lemma A.17.

$\langle 4\rangle 2.$ Q.E.D.

Proof: From step $\langle 4\rangle 1$, along with assumption $\langle 0\rangle{:}2$ and case assumption $\langle 3\rangle$.

$\langle 3\rangle 3.$ Case: $G_0 = G' \setminus \{e\}$ and $e \notin G_3.\mathsf{E}$ and $\exists u' \in G_3.\mathsf{E}$ such that $\langle w, u'\rangle \in G_3.\mathtt{rf}$

Proof: By choosing $G_1' = G_3$, since $e \notin G_3$ and no revisiting is performed (assumptions $\langle 3\rangle$ and $\langle 2\rangle 2.1$), and also $G = G_0$ (assumptions $\langle 0\rangle{:}2$ and $\langle 3\rangle$), it trivially is $G \Rightarrow_{nr}{}^{m} G_1'$.

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

Proof: From $\langle 2\rangle 2.3$ we get that cases $\langle 3\rangle 2$ and $\langle 3\rangle 3$ are exhaustive.

$\langle 2\rangle 4.$ There is at least one G_p such that $G_1' \overset{e}{\Rightarrow}_{nr} G_p$

Proof: From Prop. A.2 and steps $\langle 2\rangle 3$ and $\langle 2\rangle 2.2$.

$\langle 2\rangle 5.$ It is $a \in T$ when G_1' is reached in the production $G \overset{a}{\Rightarrow}_{nr} G_1 \Rightarrow_{nr}{}^{*} G_1'$

Proof: From $\langle 2\rangle 3$ and $\langle 1\rangle 1$. Note that, if there are more than one alternatives for the first step of the production $G \overset{a}{\Rightarrow}_{nr} G_1 \Rightarrow_{nr}{}^{m-1} G_1'$, it will be $a \in T$ in one of them.

$\langle 2\rangle 6.$ It is $\mathsf{cons}(G_2)$

Proof: From assumption $\langle 0\rangle{:}2$.

$\langle 2\rangle 7.$ We can choose G_p such that either $\langle a, e\rangle \notin G_p.\mathtt{porf}$ or $w \in W_{excl}$ when e is added

$\langle 3\rangle 1.$ Case: $\nexists r \in G_1'.\mathtt{rf}$ such that $\langle e, r\rangle \in G_1'.\mathtt{rf}$

$\langle 4\rangle 1.$ There exists G_p such that $\langle w, e\rangle \in G_p.\mathtt{rf}$

Proof: From steps $\langle 2\rangle 2$ and $\langle 2\rangle 4$, as well as case assumption $\langle 3\rangle$, we can choose $G_p = G_3$.

$\langle 4\rangle 2.$ Q.E.D.

Proof: From case assumption $\langle 3\rangle$, and steps $\langle 2\rangle 4$ and $\langle 4\rangle 1$, it cannot be $\langle a, e\rangle \in G_p.\mathtt{porf}$.

$\langle 3\rangle 2.$ Case: $e \notin G_1'.\mathsf{E}$ and $\exists u' \in G_1'.\mathsf{E}$ such that $\langle w, u'\rangle \in G_1'.\mathtt{rf}$

Proof: From case assumption $\langle 3\rangle$ we get that it will be $w \in W_{excl}$ for any G_p.

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

Proof: From $\langle 2\rangle 3$ we get that cases $\langle 3\rangle 1$ and $\langle 3\rangle 2$ are exhaustive.

$\langle 2\rangle 8.$ Q.E.D.

$\langle 1\rangle 5.$ Q.E.D.

Proof: From $\langle 0\rangle{:}4$ we conclude that cases $\langle 1\rangle 3$ and $\langle 1\rangle 4$ are exhaustive.

□

Proposition A.16 (Non-sbrf extension). *If $\varnothing \rightarrow^{*} G \xrightarrow{e} G_1$, with $\mathtt{typ}(e) = \mathtt{RMW}$ and $\langle w, e\rangle \in G_1.\mathtt{rf}$ for some $w \in G_1.\mathsf{E}$, then $\forall n \in \mathbb{N}$ such that $G_1 \Rightarrow_{nr}{}^{n} G_1'$ it is $\exists m, G_0, G_2, G_3$ such that*

1. $G_0 \Rightarrow_{nr}{}^{m} G_2$

2. One of the following holds

a. $m = n$

b. $m < n$ and exists $a \in \mathsf{E}$ such that $G_2 \overset{a}{\Rightarrow}_{nr} G_3 \land \mathtt{tid}(a) = \mathtt{tid}(e)$

3. One of the following holds

a. $G_0 = G_1$ and $\nexists r \in G_2.\mathsf{E}$ such that $\langle e, r\rangle \in G_2.\mathtt{rf}$

b. $G_0 = G_1 \setminus \{e\}$ and $e \notin G_2.\mathsf{E}$ and $\exists u' \in G_2.\mathsf{E}$ such that $\langle w, u'\rangle \in G_2.\mathtt{rf}$

Proof sketch: We prove the argument inductively and use the graph extensibility properties to guarantee that the claims hold at each step.

Assume: 1. P is a deterministic and finite program

2. $\varnothing \rightarrow^{*} G \xrightarrow{e} G_1$

3. $\mathtt{typ}(e) = \mathtt{RMW}$

4. $\langle w, e\rangle \in G_1.\mathtt{rf}$ for some $w \in G_1.\mathsf{E}$

5. There exists $n \in \mathbb{N}$ such that $G_1 \Rightarrow_{nr}{}^{n} G_1'$

Prove: There exist m'', G_0'', G_2'', G_3'' such that

1. $G_0'' \Rightarrow_{nr}{}^{m}\ G_2''$
2. One of the following holds
   a. $m'' = n$
   b. $m'' < n$ and exists $a'' \in \mathsf{E}$ such that $G_2 \overset{a''}{\Longrightarrow}_{nr} G_3 \land \mathtt{tid}(a'') = \mathtt{tid}(e)$
3. One of the following holds
   a. $G_0'' = G_1$ and $\nexists r \in G_2''.\mathsf{E}$ such that $\langle e, r\rangle \in G_2''.\mathtt{rf}$
   b. $G_0'' = G_1 \setminus \{e\}$ and $e \notin G_2''.\mathsf{E}$ and $\exists u' \in G_2''.\mathsf{E}$ such that $\langle w, u'\rangle \in G_2''.\mathtt{rf}$

⟨1⟩1. Case: $n = 0$

Proof: Obviously it is $G_1 = G_1'$. By choosing $m'' = 0$ and $G_0'' = G_1$, it is $m'' = n$, and $\nexists r \in G_1.\mathsf{E}$ such that $\langle e, r\rangle \in G_1.\mathtt{rf}$ (assumption ⟨0⟩:2).

⟨1⟩2. Case: 1. $n = k + 1$
2. There exist m', G_0', G_2', G_3' such that
   a. $G_0' \Rightarrow_{nr}{}^{m'}\ G_2'$
   b. One of the following holds
      i. $m' = k$
      ii. $m' < k$ and exists $a' \in \mathsf{E}$ such that $G_2' \overset{a'}{\Longrightarrow}_{nr} G_3' \land \mathtt{tid}(a') = \mathtt{tid}(e)$
   c. One of the following holds
      i. $G_0' = G_1$ and $\nexists r \in G_2'.\mathsf{E}$ such that $\langle e, r\rangle \in G_2'.\mathtt{rf}$
      ii. $G_0' = G_1 \setminus \{e\}$ and $e \notin G_2'.\mathsf{E}$ and $\exists u' \in G_2'.\mathsf{E}$ such that $\langle w, u'\rangle \in G_2'.\mathtt{rf}$

⟨2⟩1. Case: $m' = k$

⟨3⟩1. Choose G_3' such that $G_2' \overset{e'}{\Longrightarrow}_{nr} G_3'$

Proof: From assumptions ⟨0⟩:5, ⟨1⟩:1, and graph extensibility we get that Algorithm 1 can indeed take a step.

⟨3⟩2. Case: $\mathtt{tid}(e') = \mathtt{tid}(e)$

Proof: By choosing $m'' = k, G_0'' = G_0', G_2'' = G_2', G_3'' = G_3'$ we get that

1. $m'' < k + 1 = n$ and for $a'' = e'$ it is $G_2'' \overset{a''}{\Longrightarrow}_{nr} G_3'' \land \mathtt{tid}(a'') = \mathtt{tid}(e)$ (assumptions ⟨1⟩:1, ⟨2⟩, and ⟨3⟩)
2. One of the following holds (from assumption ⟨1⟩:2c)
   a. $G_0'' = G_1$ and $\nexists r \in G_2''.\mathsf{E}$ such that $\langle e, r\rangle \in G_2''.\mathtt{rf}$
   b. $G_0'' = G_1 \setminus \{e\}$ and $e \notin G_2''.\mathsf{E}$ and $\exists u' \in G_2''.\mathsf{E}$ such that $\langle w, u'\rangle \in G_2''.\mathtt{rf}$

⟨3⟩3. Case: $\mathtt{tid}(e') \neq \mathtt{tid}(e)$

⟨4⟩1. Case: $G_0' = G_1$ and $\nexists r \in G_2'.\mathsf{E}$ such that $\langle e, r\rangle \in G_2'.\mathtt{rf}$

⟨5⟩1. Case: $\mathtt{typ}(e') = \mathtt{W}$

Proof: By choosing $m'' = k + 1, G_0'' = G_0', G_2'' = G_3'$ we get

1. $G_0'' \Rightarrow_{nr}{}^{m''}\ G_2''$ (assumptions ⟨1⟩:2a, ⟨3⟩, and step ⟨3⟩1)
2. $m'' = n$ (assumption ⟨1⟩:1)
3. $G_0'' = G_1$ and $\nexists r \in G_2''.\mathsf{E}$ such that $\langle e, r\rangle \in G_2''.\mathtt{rf}$ (step ⟨3⟩1, $e' \notin \mathit{rng}(G_2''.\mathtt{rf})$ because of ⟨5⟩, and case assumptions ⟨3⟩ and ⟨4⟩)

⟨5⟩2. Case: $\mathtt{typ}(e') = \mathtt{R}$

Proof: By choosing $m'' = k + 1, G_0'' = G_0', G_2'' = G_3'$ we get

1. $G_0'' \Rightarrow_{nr}{}^{m''}\ G_2''$ (assumptions ⟨1⟩:2a, ⟨3⟩, and step ⟨3⟩1)
2. $m'' = n$ (assumption ⟨1⟩:1)
3. $G_0'' = G_1$ and $\nexists r \in G_2''.\mathsf{E}$ such that $\langle e, r\rangle \in G_2''.\mathtt{rf}$ (from step ⟨3⟩1, case assumption ⟨5⟩ and Prop. A.4–Item (a), as well as case assumptions ⟨3⟩ and ⟨4⟩)

⟨5⟩3. Case: $\mathtt{typ}(e') = \mathtt{RMW}$

⟨6⟩1. Exists $w' \in G_3'.\mathsf{E}$ such that $\langle w', e'\rangle \in G_3'.\mathtt{rf}$

Proof: From $\langle 3\rangle 1$, case assumption $\langle 5\rangle$, and graph extensibility.

$\langle 6\rangle 2$. Case: $w' \neq e$

Proof: By choosing $m'' = k + 1, G_0'' = G_0', G_2'' = G_3'$ we get

1. $G_0'' \Rightarrow_{nr}{}^{m''} G_2''$ (assumptions $\langle 1\rangle$:2a, $\langle 3\rangle$, and step $\langle 3\rangle 1$)
2. $m'' = n$ (assumption $\langle 1\rangle$:1)
3. $G_0'' = G_1$ and $\nexists r \in G_2''.\mathsf{E}$ such that $\langle e, r\rangle \in G_2''.\mathtt{rf}$ (from steps $\langle 3\rangle 1$ and $\langle 6\rangle 2$, as well as case assumptions $\langle 3\rangle$ and $\langle 4\rangle$)

$\langle 6\rangle 3$. Case: $w' = e$

$\langle 7\rangle 1$. Let $G_2'' \triangleq \mathsf{SetRF}(G_3' \setminus \{e\}, w, e')$ such that $G_0' \Rightarrow_{nr}{}^{k+1} G_2''$ and $\exists u' \in G_2''.\mathsf{E}$ such that $\langle w, u'\rangle \in G_2''.\mathtt{rf}$

Proof: From $\langle 1\rangle$:2a we know that $G_0' \Rightarrow_{nr}{}^{k} G_2'$. Then, from graph extensibility (Prop. A.4–Item (b)) with case assumptions $\langle 6\rangle$, $\langle 5\rangle$, $\langle 3\rangle$, and assumptions $\langle 0\rangle$:3 and $\langle 0\rangle$:4 we get the desired result, with $u' = e'$.

$\langle 7\rangle 2$. Q.E.D.

Proof: By choosing $m'' = k + 1, G_0'' = G_0' \setminus \{e\}$ and G_2'' from $\langle 7\rangle 1$ we get

1. $G_0'' \Rightarrow_{nr}{}^{m''} G_2''$ (from Lemma A.17 with step $\langle 7\rangle 1$, and case assumptions $\langle 4\rangle$ and $\langle 1\rangle$:2a)
2. $m'' = n$ (assumption $\langle 1\rangle$:1)
3. $G_0'' = G_1 \setminus \{e\}$ and $e \notin G_2'.\mathsf{E}$ and $\exists u' \in G_2''.\mathsf{E}$ such that $\langle w, u'\rangle \in G_2''.\mathtt{rf}$ (from case assumption $\langle 4\rangle$ and step $\langle 7\rangle 1$)

$\langle 6\rangle 4$. Q.E.D.

Proof: Cases $\langle 6\rangle 2$ and $\langle 6\rangle 3$ are exhaustive.

$\langle 5\rangle 4$. Q.E.D.

Proof: Cases $\langle 5\rangle 1$-$\langle 5\rangle 3$ are exhaustive.

$\langle 4\rangle 2$. Case: $G_0' = G_1 \setminus \{e\}$ and $e \notin G_2'.\mathsf{E}$ and $\exists u' \in G_2'.\mathsf{E}$ such that $\langle w, u'\rangle \in G_2'.\mathtt{rf}$

Proof: By choosing $m'' = k + 1, G_0'' = G_0, G_2'' = G_3'$ we get

1. $G_0'' \Rightarrow_{nr}{}^{m''} G_2''$ (case assumptions $\langle 1\rangle$:2a, $\langle 3\rangle$, and step $\langle 3\rangle 1$)
2. $m'' = n$ (assumption $\langle 1\rangle$:1)
3. $G_0'' = G_1 \setminus \{e\}$ and $e \notin G_2''.\mathsf{E}$ and $\exists u' \in G_2''.\mathsf{E}$ such that $\langle w, u'\rangle \in G_2''.\mathtt{rf}$ (from graph extensibility, step $\langle 3\rangle 1$, and case assumptions $\langle 3\rangle$ and $\langle 4\rangle$)

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

Proof: From $\langle 1\rangle$:2c we get that cases $\langle 4\rangle 1$ and $\langle 4\rangle 2$ are exhaustive.

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

Proof: Cases $\langle 3\rangle 2$ and $\langle 3\rangle 3$ are exhaustive.

$\langle 2\rangle 2$. Case: $m' < k$ and exists $a' \in \mathsf{E}$ such that $G_2' \overset{a'}{\Longrightarrow}_{nr} G_3' \land \mathtt{tid}(a') = \mathtt{tid}(e)$

Proof: By choosing $m'' = m'$, $G_0'' = G_0'$, $G_2'' = G_2'$, $G_3'' = G_3'$, and $a'' = a'$, we get that

1. $m'' < k + 1 = n$ and exists $a'' \in \mathsf{E}$ such that $G_2'' \overset{a''}{\Longrightarrow}_{nr} G_3'' \land \mathtt{tid}(a'') = \mathtt{tid}(e)$ (assumptions $\langle 1\rangle$:1 and $\langle 2\rangle$)
2. One of the following holds (from assumption $\langle 1\rangle$:2c)
   a. $G_0'' = G_1$ and $\nexists r \in G_2''.\mathsf{E}$ such that $\langle e, r\rangle \in G_2''.\mathtt{rf}$
   b. $G_0'' = G_1 \setminus \{e\}$ and $e \notin G_2''.\mathsf{E}$ and $\exists u' \in G_2''.\mathsf{E}$ such that $\langle w, u'\rangle \in G_2''.\mathtt{rf}$

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

Proof: Cases $\langle 2\rangle 1$ and $\langle 2\rangle 2$ are exhaustive (assumption $\langle 1\rangle$:2b).

$\langle 1\rangle 3$. Q.E.D.

Proof: Steps $\langle 1\rangle 1$, $\langle 1\rangle 2$ and mathematical induction.

□

Lemma A.17 (Equivalent production). *If there are* $n \in \mathbb{N}, G_0, G_n, e \in \mathsf{E}$ *such that* $\mathsf{cons}(G_0)$, $G_0 \Rightarrow_{nr}{}^n G_n$ *and for all* $G_i \in [G_0 \Rightarrow_{nr}{}^n G_n]$ *it is* $\mathsf{nextEvent}_P(G_i) \neq e$ *and either* $rng([e]; G_i.\mathsf{porf}) = \emptyset$ *or* $e \notin G_i.\mathsf{E}$*, then there is an equivalent production* $G_0 \setminus \{e\} \Rightarrow_{nr}{}^n G_n \setminus \{e\}$.

Proof sketch: Simple proof by induction.

Assume: 1. P is a deterministic program with finite traces
2. $n \in \mathbb{N}$, $e \in \mathsf{E}$, and G_0 such that $\mathsf{cons}(G_0)$
3. $G_0 \Rightarrow_{nr}{}^n G_n$ such that, for all $G_i \in [G_0 \Rightarrow_{nr}{}^n G_n]$, it is
a. $\mathsf{nextEvent}_P(G_i) \neq e$
b. $rng([e]; G_i.\mathsf{porf}) = \emptyset$ or $e \notin G_i.\mathsf{E}$

Prove: $G_0 \setminus \{e\} \Rightarrow_{nr}{}^n G_n \setminus \{e\}$ and for all $G'_i \in [G_0 \setminus \{e\} \Rightarrow_{nr}{}^n G_n \setminus \{e\}]$ it is $e \notin G'_i$

⟨1⟩1. Case: $n = 0$
Proof: It obviously is $G_0 = G_n$ (assumption 3). From prefix-closedness (Def. A.6) and assumption 2, it also is $\mathsf{cons}(G_0 \setminus \{e\})$.

⟨1⟩2. Case: 1. $n = k + 1$
2. *IH*: Given a G'_k such that $G_0 \Rightarrow_{nr}{}^k G'_k$, where for all $G'_i \in [G_0 \Rightarrow_{nr}{}^k G'_k]$ it is
a. $\mathsf{nextEvent}_P(G'_i) \neq e$
b. $rng([e]; G'_i.\mathsf{porf}) = \emptyset$ or $e \notin G'_i.\mathsf{E}$
then it also is $G_0 \setminus \{e\} \Rightarrow_{nr}{}^k G'_k \setminus \{e\}$

⟨2⟩1. Let G_k, G_{k+1} such that $G_0 \Rightarrow_{nr}{}^k G_k \overset{e'}{\Rightarrow}_{nr} G_{k+1}$ and $\forall G_i \in [G_0 \Rightarrow_{nr}{}^k G_k \overset{e'}{\Rightarrow}_{nr} G_{k+1}]$ it is
1. $\mathsf{nextEvent}_P(G_i) \neq e$
2. $rng([e]; G_i.\mathsf{porf}) = \emptyset$ or $e \notin G_i.\mathsf{E}$
Proof: From assumptions ⟨0⟩:3 and ⟨1⟩:1.

⟨2⟩2. Case: $e \notin G_{k+1}$
Proof: It is $G_{k+1} \setminus \{e\} = G_{k+1}$ (case assumption ⟨2⟩), and since from the *IH* we get that $G_0 \setminus \{e\} \Rightarrow_{nr}{}^k G_k \setminus \{e\}$ (assumption ⟨1⟩:2) with $\mathsf{cons}(G_k \setminus \{e\})$ (courtesy of Algorithm 1), it also is $G_0 \setminus \{e\} \Rightarrow_{nr}{}^k G_k \setminus \{e\} \overset{e'}{\Rightarrow}_{nr} G_{k+1} \setminus \{e\}$ (from step ⟨2⟩1).

⟨2⟩3. Case: $rng([e]; G_{k+1}.\mathsf{porf}) = \emptyset$

⟨3⟩1. It is $e' \neq e$
Proof: From ⟨2⟩1.

⟨3⟩2. Q.E.D.
Proof: From prefix-closedness (Def. A.6) we get that it is $\mathsf{cons}(G_{k+1} \setminus \{e\})$, and from the *IH* (assumption ⟨1⟩:2), we get that $G_0 \setminus \{e\} \Rightarrow_{nr}{}^k G_k \setminus \{e\}$ with $\mathsf{cons}(G_k \setminus \{k\})$ (algorithmic construction). Thus, from step ⟨3⟩1, it can also be $G_0 \setminus \{e\} \Rightarrow_{nr}{}^{k+1} G_{k+1} \setminus \{e\}$.

⟨2⟩4. Q.E.D.
Proof: From assumption ⟨0⟩:3b and step ⟨2⟩1 we get that cases ⟨2⟩2 and ⟨2⟩3 are exhaustive.

⟨1⟩3. Q.E.D.
Proof: Steps ⟨1⟩1, ⟨1⟩2 and mathematical induction.

□