Model Checking for Weakly Consistent Libraries

Michalis Kokologiannakis  
MPI-SWS  
michalis@mpi-sws.org

Azalea Raad  
MPI-SWS  
azalea@mpi-sws.org

Viktor Vafeiadis  
MPI-SWS  
viktor@mpi-sws.org

Abstract

We present GenMC, a model checking algorithm for concurrent programs that is parametric in the choice of memory model and can be used for verifying clients of concurrent libraries. Subject to a few basic conditions about the memory model, our algorithm is sound, complete and optimal, in that it explores each consistent execution of the program according to the model exactly once, and does not explore inconsistent executions or embark on futile exploration paths. We implement GenMC as a tool for verifying C programs. Despite the generality of the algorithm, its performance is comparable to the state-of-art specialized model checkers for specific memory models, and in certain cases exponentially faster thanks to its coarse partitioning of executions.

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

Keywords  Model checking, weak memory models

ACM Reference Format:

1 Introduction

Suppose that we have a concurrent program, e.g.,

\[
\begin{align*}
x & := 1 \\
y & := 1 \
\end{align*}
\]

\[
\begin{align*}
a & := y \\
b & := x \\
\text{assert}(a \leq b)
\end{align*}
\]

with \(x\) and \(y\) initialized with 0, and we want to check whether its assertions are always satisfied. An effective way of doing so is using stateless model checking (SMC) [18, 19, 34], which enumerates all executions of the program and checks each execution individually. SMC has two major challenges.

The first challenge is associated with the memory model under which the program is executed, as it determines the program outcomes. For example, in the MP program above, the assertion \((a \leq b)\) holds under SC [28] and TSO [36], but not under PSO [40], or RC11 with \('relaxed'\) accesses [27], because the latter two models allow for writes to distinct locations to be reordered. Thus, under PSO and RC11, \(y := 1\) may execute before \(x := 1\), thereby violating the assertion.

The second challenge is that any non-trivial concurrent program has a large number of executions that need to be explored (typically, exponential in the size of the program). To tackle this, partial order reduction techniques [1, 12, 16, 20, 42] have been developed, and try to partition the executions into equivalence classes and explore exactly one execution per equivalence class.

However, while there exist efficient techniques that target specific memory models [1–4, 12, 15, 16, 20–22, 26, 35, 39, 42], a generic technique that combats both these challenges is yet to be developed.

The goal of this paper is to develop such a model checking algorithm that is parametric in the choice of the memory model. Our algorithm, GenMC (Generic Model Checker), can be used not only for traditional memory models supporting reads, writes, and read-modify-write (RMW) instructions, but also for models incorporating high-level libraries, such as mutual exclusion locks, as primitive operations.

Our contributions can be summarized as follows:

- Through a series of examples, we present an intuitive account of our algorithm for verifying concurrent programs, using execution graphs and axiomatic semantics for any memory model (§2), so long as it satisfies four basic assumptions: porf-acyclicity, extensibility, prefix-closedness and well-blocking (§3).
- Our approach distinguishes executions based solely on the program-order and reads-from relations (§2.5), which can lead to exponentially fewer explorations compared to approaches that maintain a total coherence order between conflicting writes (§6.3).
- We demonstrate how our technique can verify programs under memory models that incorporate high-level libraries, such as mutual exclusion locks (§2.7).
- We describe our algorithm in detail (§4), and prove that it is (a) sound: produces no false positives; (b) complete: explores all possible program behaviours; and (c) optimal: explores each behaviour exactly once.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ACM ISBN 978-1-4503-6712-7/19/06...

https://doi.org/10.1145/3314221.3314609
• We implement GenMC into a tool for verifying C programs (§5), and demonstrate that it has comparable or better performance than the state-of-the-art specialized tools for specific memory models (§6).

2 Overview

In the literature of axiomatic memory models [6, 27], the traces of shared memory accesses generated by a program are represented as a set of execution graphs, where each graph G comprises: (i) a set of events (graph nodes); and (ii) a few relations on events (graph edges). The two kinds of edges present in all memory models are the program order (po) and the reads-from relation (rf), which relates each read event r in G to a write event w in G, from which r obtains its value. The semantics of a program P is then given by the set of executions that satisfy a certain consistency predicate.

For example, sequential consistency (SC) [28] requires the existence of a total order on all events extending the program order such that each read reads from the most recent prior write to same location in that total order. Equivalently, SC can be defined in terms of a modification order, mo, also known as the coherence order. An execution is SC-consistent if there exists mo such that for every location x, mo totally orders all writes to x and po∪rf∪mo∪(rf⁻¹); mo is acyclic, where rf⁻¹ denotes the inverse of rf, and ’;’ denotes relational composition: (r, w) ∈ rf⁻¹; mo ≡ ∃w′. (w′, r) ∈ rf ∧ (w′, w) ∈ mo.

Other models are weaker and deem more executions consistent. TSO [36] allows loads to execute before po-earlier stores, while PSO [40] further allows stores of a thread to execute out of order. RC11 [27] supports various access modes ranging from SC to ones even weaker than what PSO offers.

Let us now consider a simple example program:\footnote{In all our examples, we use x, y, z as global (shared) variables and a, b, c as local variables. All variables are implicitly initialized to 0.}

\[
\begin{align*}
x & := 1 \quad \text{if } a = 0 \text{ then } y := 1 \quad x := 2 \quad (\text{w+rw+w})
\end{align*}
\]

Under SC, as depicted by the executions in Fig. 1, the read in thread 2 may read either 0 (from the initialization write), 1 (from the write in thread 1), or 2 (from thread 3).

Our goal is to enumerate such executions systematically. A simple approach taken, e.g., by Herd [6] andCppMem [8], is to enumerate all possible executions and filter them according to the consistency predicate of the memory model.

To do this, we require that the underlying memory model satisfy the following (see §3):

**MM1:** porf is irreflexive, where porf ⊆ (po ∪ rf)⁺

This requirement is satisfied by several models (e.g., SC, TSO, PSO, and RC11), and ensures that loop-free programs have finitely many executions. Without this requirement, we can easily run into problems as the following program illustrates:

\[x := y \quad y := x \quad \text{(lb+def)}\]

Under the (arguably useless) memory model that deems every execution graph consistent, the program can return \(x = y = v\), for any value \(v\), by having both threads read \(v\) and write \(v\) in a circular fashion as shown below:

\[
\begin{align*}
\text{R}(y) & \quad \text{R}(x) \quad \text{R}(y) \\
\text{W}(x, v) & \quad \text{W}(x, v) \quad \text{W}(y, v)
\end{align*}
\]

In the weak memory literature, such executions are considered problematic because they generate values “out of thin air” (OOTA) [10, 31, 41] and inhibit compositional reasoning.

**Remark 1.** While restricting OOTA behaviours, MM1 also precludes models allowing the outcome \(a = b = 1\) for the following “load buffering” litmus test:

\[
\begin{align*}
a & := y; \quad b := x; \quad x := 1 \quad y := 1 \quad \text{(lb)}
\end{align*}
\]

A few models allow this outcome and yet avoid OOTA executions. The Power [6] and ARM [37] models record (syntactic) dependencies in executions and forbid dependency cycles, while the Promising [23] and WeakestMO [11] models are not even defined in terms of execution graphs. Handling these models is beyond the scope of this paper.

2.1 Checking Consistency at Every Step

Even without OOTA executions, generating all executions and then checking consistency does not scale [24].

A much better approach, followed by most tools (e.g., [1, 2, 4, 24, 35]), is to construct executions incrementally by adding events one at a time and checking for consistency after each step. For instance, execution 2 in Fig. 1 can be generated by adding its events in the following order: \(W(x, 0), W(x, 1), R(x), \) and \(W(x, 2)\).

2.2 Fixing the Graph Construction Order

To generate all executions of a program following the condition of §2.1 one must in principle consider all possible
extensions of \textit{porf}. This, however, very often leads to duplicate explorations.

Therefore, ideally, one would generate all executions without considering all possible extensions of \textit{porf}, regardless of the memory model. In fact, we can do this for all well-known memory models. In particular, models such as SC, TSO, PSO, and RC11 all satisfy an even stronger guarantee, namely \textit{prefix-closeness}:

**MM2**: There exists a partial order \( R \) that includes reads-from and (preserved) program order such that, if a graph is consistent, so is every \( R \)-prefix of it.

This ensures that to generate a particular execution, it is sufficient to consider \textit{any} total extension of \textit{porf}.

As we demonstrate below, we can leverage this fact and fix an order in which we add execution events one at a time, thus generating all executions of a program systematically.

### 2.3 GenMC: A First Example

Let us run our model checking algorithm, \textit{GenMC}, to generate the executions of \( W+RW+W \) by adding its events in a \textit{fixed} order given by thread identifiers: first the events of (the left-most) thread 1, then the events of thread 2, and so forth.

We start with an initial graph \( G_0 \) containing only the initialization write \( W(x,0) \) (see below). First, we add the \( W(x,1) \) write of the first thread to \( G_0 \), simultaneously adding the appropriate \textit{po} edge between the events:

\[
\begin{align*}
W(x,0) & \sim W(x,1) \\
W(x,0) & \sim W(x,0)
\end{align*}
\]

Continuing in thread order, we next add the \( R(x) \) read of thread 2, which may read from either of the writes in the graph, yielding two distinct graphs (one for each case):

\[
\begin{align*}
W(x,0) & \sim \begin{cases} \\
W(x,0) & W(x,1) \sim R(x) \\
W(x,0) & W(x,1) \sim R(x)
\end{cases}
\end{align*}
\]

However, recording both graphs is inefficient: in the general case, we need to record one graph for each of the reads-from options of each read. Note that the two graphs are identical up to the read, which is the point of divergence. As such, each time we add a read that can read from more than one place, we proceed with one of the options, e.g., \( W(x,0) \), and record the alternative(s), i.e., \( W(x,1) \), into a work list \( W \) for later exploration. \( W \) maps each read to a list of writes it can also read from; in this case, the current graph along with \( W \) is given below. We refer to revisit options such as \( W(x,1) \) as \textit{forward} revisits since they are already in the graph when the read \( R(x) \) is added to the graph.

\[
\begin{align*}
W(x,0) & \sim W(x,1) \\
W(x,0) & \sim W(x,0)
\end{align*}
\]

Since the value read is 0, we next add the \( W(y,1) \) write of thread 2. Finally, we add \( W(x,2) \) of thread 3 which yields the graph below. Note that, as it is consistent for the read to read 2 from this newly added write, we also record this new reads-from as a revisit option in \( W \). We refer to revisit options such as \( W(x,2) \) as \textit{backward} revisits since they are added to the graph after the corresponding read \( R(x) \).

This first execution is now completed (denoted by the highlighted background): it corresponds to execution \( \mathcal{C} \) of Fig. 1. To generate the remaining executions, we \textit{revisit} the graph by picking an alternative reads-from option from \( W \).

Suppose that we next pick \( W(x,2) \) from \( W \). To continue, we restrict the graph to contain only the events added to the graph \textit{prior} to (and including) the read (i.e., \( W(x,0), W(x,1) \) and \( R(x) \)), as well as the events that led up to (in \textit{porf} order) the revisiting write \( W(x,2) \). This yields the complete graph below, corresponding to execution \( \mathcal{E} \) in Fig. 1. The \( W(x,2) \) option is marked as \( \checkmark \) to denote that it has been considered.

When considering the alternative reads-from option \( W(x,2) \), we restrict the graph to contain only the events added \textit{prior} to the read. Restricting the graph is important because events added after the read may depend on its value. For instance, it is crucial to remove \( W(y,1) \) as it is only present when 0 is read from \( x \). Similarly, we must retain the events added before (in \textit{porf} order) the alternative reads-from option \( W(x,2) \). For instance, if \( x := 2 \) in \( W+RW+W \) is wrapped in the conditional if \( y = 1 \) then, the presence of the \( W(x,2) \) event in the graph depends on the value read for \( y \), i.e., the events before \( W(x,2) \) in \textit{porf} order.

To generate the last execution, we revisit the graph once again by picking the remaining option \( W(x,1) \) in \( W \). We then restrict the graph as before, yielding the graph below:

To continue, we add the \( W(x,2) \) write arriving at the graph below, corresponding to execution \( \mathcal{E} \) in Fig. 1. Note that we do not re-add this write as an entry in \( W \), as this option has already been explored (\( \checkmark \)-marked).

...
Finally, as the graph is complete, and all options in \( W \) are explored, the algorithm terminates.

**Avoiding Duplication** When revisiting a read event, write events may be removed from the graph and later re-added. As such, additional care is required to avoid duplicate backward revisits. For instance, continuing from (Ex2), by picking the next option in \( W (W (x, 1)) \), we removed \( W (x, 2) \) arriving at (Pre3). We later re-added \( W (x, 2) \) and obtained (Ex3). In doing so, we did not re-add \( W (x, 2) \) as a (backward) revisit option to \( W \) as this option had already been explored before. Rather, by having previously marked \( W (x, 2) \) as explored (\( \checkmark \)-marked), we ascertained that \( W (x, 2) \) is indeed a duplicate revisit. To this end, as we describe in §4, backward options are not removed from \( W \); instead they are marked as explored (e.g., in (Pre3) and (Ex3)). By contrast, forward revisits do not lead to duplication. This is because when revisiting a read (e.g., \( R (x) \)), only events added after the read are removed from the graph. As such, since a forward option (e.g., \( W (x, 1) \)) is added to the graph before the read, it is not removed from the graph, and therefore not re-added, avoiding duplication. For efficiency, we thus remove forward options from \( W \) once explored (e.g., \( W (x, 1) \) is removed in (Pre3) and (Ex3)).

### 2.4 GenMC: Extensible Memory Models

Note that as described in §2.3, GenMC generates all executions, even though it does not add events in \( \text{po}\text{rf} \) order. This is because in cases where a read is added before the write it reads from, e.g., reading from \( W (x, 2) \) in \( 2 \), the \( \text{rf} \) edge is recorded as an option in \( W \) once the write is added.

This then leads to the question, could events added after a read affect the consistency of the execution in a way that the write is never added and hence the alternative \( \text{rf} \) option is never considered? Perhaps surprisingly, the answer is yes. For example, consider the following program under a (contrived) memory model that dictates "if a read of \( y \) reads 0, then there cannot be a read of \( x \) that also reads 0":

\[
a := x \parallel b := y \parallel x := 42 \quad (\text{R+R+W})
\]

In this case, adding the events in thread order results in a graph where both \( x \) and \( y \) read 0, which is then dropped as inconsistent, and thus we cannot generate the execution where the first thread reads 42. This brings us to our third requirement on memory models, extensibility:

**MM3**: Given a consistent execution \( G \), a \( \text{po}\text{max} \)-maximal event can always be added to \( G \) to yield a consistent execution (with an appropriate \( \text{rf} \) edge when applicable).

This requirement holds for all well-known memory models, and excludes "nonsensical" memory models such as that above. In particular, under that model, the consistent execution of \( \text{R+R+W} \) comprising the initialization events and \( R (x) \) of the first thread reading 0 cannot be extended by adding \( R (y) \) for any choice of \( \text{rf} \).

2.5 **GenMC: Modification Order and Writes-Before**

Recall that using GenMC, we generated all three executions of \( \text{w+rw+w} \) under SC in Fig. 1. These executions, however, do not exactly correspond to the notion of executions in the formal definition of SC: as discussed above, SC executions additionally record the modification order \( \text{mo} \), which totally orders all writes to a given memory location. We refer to such execution (which record \( \text{mo} \)) as \( \text{mo}-\text{executions} \).

As such, the three executions in Fig. 1 correspond to the six \( \text{mo}\)-executions depicted in Fig. 2. In this program, each execution corresponds to two \( \text{mo}\)-executions representing the two ways \( W (x, 1) \) and \( W (x, 2) \) could be ordered by \( \text{mo} \).

One can of course adapt GenMC to enumerate all \( \text{mo}\)-executions, as e.g., in [24]; but doing so is wasteful because while the choice of \( \text{mo} \) can affect the consistency of an execution, it is not directly observable by the program. As long as checking for consistency is reasonably efficient, enumerating only (plain) executions is better because it searches through a space that is up to exponentially smaller.²

Now, how can we check consistency of an execution besides naively enumerating all \( \text{mo} \) possibilities? The idea is to compute the "writes-before" (\( \text{wb} \)) relation, which records the set of \( \text{mo}\)-edges whose direction is forced because of the \( \text{rf} \)-edges. Let us consider the following executions under SC:

²To see that, consider an extension of \( \text{w+rw+w} \) with \( n \) parallel writes and one reader: that program has \( n + 1 \) executions and \( (n + 1)! \) \( \text{mo}\)-executions.
We next pick the alternative option for $b$ from $W$, restrict the graph as before, and obtain the (inconsistent) execution below where both RMWs read $0$. Additionally, we check whether the read being revisited (i.e., $b$) may itself generate backward revisit options for existing reads in the graph. In this case, $a$ can read from $b$ and thus $b$ is added as a revisit option for $a$. This graph is then dropped as it is inconsistent (violates RMW atomicity), as denoted by the lined-background.

Finally, we pick the remaining revisit option in $W[a]$, restrict the graph as before and arrive at execution $\mathcal{G}_3$.

2.7 GenMC: Model Checking for Libraries

We next explain how GenMC generalizes to models incorporating high-level (abstract) libraries. To do so, let us consider a mutex library with lock and unlock instructions.

Although the mutex library does not have conventional read and write operations, its primitives behave very much like reads and writes. Intuitively, unlock can be viewed as a write, while lock can be viewed as a read that may either read from an initial value (i.e., acquiring the mutex immediately after it is initialized), or read from an unlock instruction (i.e., acquiring the mutex after it has been released by its previous holder). As with RMWs, the mutex library requires rf-functionality: no two lock events read from the same place, capturing the exclusivity of the mutex while held.

An interesting feature of the mutex library is that the calls to lock may block if the mutex is taken. Put formally, when all writes (initialization and unlocks) in an execution have already been read-from, due to rf-functionality, when adding a read (lock) event $e$ to the graph, there may not exist a write from which $e$ could read. When this is the case, the read event $e$ blocks in that its thread cannot make progress and thus $e$ has no porf successors. Note that in such libraries rf is not necessarily total on reads. However, lock events may not block arbitrarily: a lock may block only when all writes are read from; i.e., when the mutex is taken. This brings us to our final requirement on memory models, well-blocking:

**MM4:** Given a consistent execution $G$: 1) blocking reads in $G$ have no porf successors; and 2) if $G$ contains a blocking read, then all writes in $G$ are read from.

Consider the program below with its executions in Fig. 4:

$$a : \text{lock}(l); \quad b : \text{lock}(l);$$

Note that neither lock call may block as the program contains sufficient writes: two unlocks and the implicit initialization.

Running our algorithm on this example, we add the events in order $(a, a', b, b')$ and obtain execution $(1)$. As with $\mathcal{G}_2$, when adding $b$ to the graph, we also consider inconsistent

---

3The definition of $\mathcal{G}b$ can be found in our technical appendix [25].
reads-from options and add them to the work list, arriving at the following configuration:

We then pick the next option for $b$ and restrict the graph as before. As in the FAI/2 example, we check whether $b$ may itself generate backward revisits options for the reads in the graph. However, since $b : lock(l)$ is only a read event (in contrast to $b : RMW(x, 2)$ in FAI/2 which is also a write), $a$ cannot read from $b$. Nonetheless, by reading from the initialization event, $b$ causes $a$ to block. That is, blocking (⊥) is added as a revisit option for $a$. This graph is subsequently dropped as inconsistent (violating rf-functionality):

We next pick $\perp$ as a revisit option for $a$. Since $a$ is now blocking, its thread cannot proceed and its subsequent events are skipped. We thus nest $\perp$ to the graph. As $b'$ is a write, it may revisit $a$ and is added as an option in $W[a]$. However, adding $b'$ renders the graph inconsistent (as blocking despite the available $b'$) and is thus dropped:

Finally, we consider the last revisit option ($b'$) for $a$. After restricting the graph, we add event $a'$ and obtain (2) in Fig. 4.

Note that running GenMC on LOCK/2 was no different from running it on FAI/2 and required no special treatment: we merely used the lock library consistency check rather than that of RC11. Indeed, the main difference between the two examples is the blocking behaviour of locks, which is prescribed by the lock library specification. As such, GenMC can be adapted to any memory model that meets the conditions in MM1-MM4. We next formalize these conditions.

3 Formal Model

We describe a framework for axiomatic memory models (MMs) and instantiate it to specify a mutex library. In the technical appendix [25], we present the SC [28], TSO [36] and RC11 [27] models as instances of this framework.

Execution Graphs The traces of a program are represented as a set of execution graphs, where each graph $G$ comprises: (i) a set of events; and (ii) a number of relations on events.

An event is a tuple of the form $(i, n, l)$, where $i \in Tid \cup \{0\}$ is a thread identifier (0 for initialization events) with $Tid \subseteq N$, $n \in N$ is the serial number inside a thread, and $l \in Lab$ 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 $error$ (denoting assertion violations); or (ii) the stuck label $stuck$ (e.g., due to a failed $assume$ statement); or (iii) a memory model-specific label, e.g., the write label $w(x, 1)$ for writing $1$ to $x$ under the SC model. The label function $lab$ returns the label of an event. We assume a set of locations Loc; the loc function returns the location of a label.

Definition 3.1 (Executions). Given designated sets of read ($R$) and write ($W$) events, an execution is a tuple $G=(E, rf)$, where $E$ is a sequence of events, and $rf : E \cap R \rightarrow E \cap W$ is the reads-from function.

The sets of read and write events are designated by the memory model and are not necessarily low-level reads/writes. For instance, in case of the mutex library, lock and unlock events constitute read and write events, respectively.

Recall from §2.2 that to generate program executions using our algorithm, it suffices to fix the construction order. This is given by the order of events in the sequence $E$.

Given an execution $G$, we write $G.E$ and $G.rf$ for its components, and write $G.R$ (resp. $G.W$) for $G.E \cap R$ (resp. $G.E \cap W$). We write $G.E_i$ for $\{(i', n', l') \in G.E \mid i = i'\}$; and write $G.po$ for the program order defined as follows:

\[
G.po = G.E_0 \times (G.E \setminus G.E_0) \cup \{(i_1, n_1, l_1), (i_2, n_2, l_2) \in G.E \setminus G.E_0 \mid l_1 = l_2 \land n_1 < n_2\}
\]

In general, $G.rf$ may not be a total function: read events that do not read from any event are used to model blocking library events, such as a blocking lock event that is awaiting the release of a mutex. We write $G.B \equiv G.R \setminus dom(G.rf)$ for the set of blocked events. Finally, although $G.rf$ is a function, we often implicitly coerce it to a relation on $W \times R$.

Notation Given a relation $r$ and a set $A$, we write $r^T$, $r^*$ and $r^*$ for the reflexive, transitive and reflexive-transitive closure of $r$, respectively. We write $dom(r)$ and $rng(r)$ for the domain and range of $r$, respectively. We write $r^{-1}$ for the inverse of $r$; $r|_A$ for $r \cap (A \times A)$; and $[A]$ for the identity relation on $A$: $\{(a, a) \mid a \in A\}$. Given relations $r_1$ and $r_2$,
we write $r_1; r_2$ for $\{(a, b) \mid \exists c. (a, c) \in r_1 \land (c, b) \in r_2\}$, i.e., their relational composition. Given an event set $E$, we write $E_x$ for $\{e \in E \mid \lnot oc(e) \land x\}$, and $G|_E$ for $(E', G.rf|_E)$ with $E' \equiv G.E \cap E$. We write $G.porf$ for $(G.po \cup G.rf)^*$, and write $G.rf(r \rightarrow w)$ for the graph obtained from mapping $G.rf(r)$ to $w$. Finally, we write $+$ for sequence concatenation.

**Extension** We define graph **extension** in Def. 3.2, used by the incremental construction in GENMC, which describes adding an available event to an execution. Given an execution $G$, an event $(i, n, l)$ is available when thread $i$ contains $n$ events, none of which are blocking. As executions are constructed incrementally by adding one available event at a time, it follows that the events of each thread $i$ are indexed with adjacent integers $1 \cdots |G.E_i|$

**Definition 3.2** (Extension). An event $e=(i, n, l)$ is available for an execution $G$ if $|G.E_i| = n - 1$ and $G.E_i \cap G.B = \emptyset$. The extension of $G$ with an available event $e$, written $Add(G, e)$, denotes the execution $(E+[e], G.rf)$.

**Consistency and Memory Model Assumptions** Given a program $P$, the admissible behaviours of $P$ are commonly described as a set of consistent executions. Consistency of an execution is memory model (MM)-specific; as such, MMs often define a consistency predicate that prescribes the conditions required for consistency. As our model checking technique is MM-parametric, we assume the existence of such a consistency predicate: given an execution $G$, we write $cons_m(G)$ to denote that $G$ is consistent under memory model $m$.

Recall from §2 that we require underlying memory models to satisfy certain properties as outlined by MM1-MM4. In what follows, we formally define these conditions.

The first condition (MM1) is captured by Def. 3.3. This well-formedness condition additionally requires that the MM be agnostic to the order in which events are added to the graph, as it constitutes auxiliary instrumentation used by our algorithm. As such, execution consistency must be independent of this order: if $(E, rf)$ is consistent then $(E', rf)$ is also consistent, where $E' \forall \in perm(E)$ is a permutation of $E$.

**Definition 3.3** (Well-formedness). An execution $G$ is well-formed if $G.porf$ is irreflexive. A memory model $m$ is well-formed iff for all $G$, if $cons_m(G)$ holds, then $G$ is well-formed, and $\forall E \in perm(G.E). cons_m((E, G.rf))$.

The prefix-closedness condition (MM2) is captured by Def. 3.4. A consistency model $m$ is commonly considered prefix-closed iff given a consistent execution $G$ and a porf-closed set of events $E \subseteq G.E$ (i.e., $dom(G.porf; [E]) \subseteq E$), restricting the graph to those events in $E$ yields a consistent execution, i.e., $cons_m(G|_E)$. However, this definition is too strong due to blocking reads.

To see this, consider the program $l_1 : lock(l) \parallel l_2 : lock(l)$. Under the mutex specification described in §2.7, one consistent execution of this program is a graph $G$ in which $l_1$ reads from mutex initialization, whilst $l_2$ blocks. Let $E = \{l_2, \text{init}\}$; if we now restrict $G$ to $E$, the resulting graph is inconsistent since $l_2$ blocks despite the available initialization event.

We thus weaken prefix-closedness by requiring that there exist a set of blocking events $B \subseteq E$ such that the graph restricted to $E \setminus B$ is consistent: $cons_m(G|_{E \setminus B})$. For instance, in the example above we can pick $B = \{l_1\}$. Note that for well-known memory models such as SC, TSO and RC11, the strong and weak notions of prefix-closedness coincide, as these models do not contain blocking events.

**Definition 3.4** (Prefix-closedness). A memory model $m$ is prefix-closed iff for all $G, E \subseteq G.E$, if $dom(G.porf; [E]) \subseteq E$ and $cons_m(G)$, then there exists $B \subseteq G.B$ such that $cons_m(G|_{E \setminus B})$.

Memory model extensibility (MM3) is captured in Def. 3.5 and requires that a memory model be read-, write- and rw-extensible. The first two requirements are intuitive and stipulate that a consistent execution can always be extended by a read or write event, respectively. The rw-extensibility imposes certain conditions on events that are both read and write events (e.g., RC11 RMW events). These requirements are rather technical and are necessary for the correctness of our algorithm (see the technical appendix [25]).

**Definition 3.5** (Extensibility). A memory model $m$ is read-extensible iff for all $G$, $r \in R$ and $G' = Add(G, r)$, if $cons_m(G)$, there exists $w \in G.w \cup \{l\}$ such that $cons_m(G'.rf[r \rightarrow w])$.

A memory model $m$ is write-extensible iff for all $G, w \in G.w$, if $cons_m(G|_{G.E \setminus \{w\}})$ and $rng([w]; G.porf) = \emptyset$, then $cons_m(G)$.

A memory model $m$ is rw-extensible iff for all $G, r, w, u$, if $cons_m(G)$, $u'$ $\in G.R \cap G.w$ and $rng([u]; G.porf) = \emptyset$, then:

- if $(u, r) \in G.rf$ and $rng([r]; G.pof) = \emptyset$, then there exist $w \in G.E \setminus \{u\}$ such that $cons(G.rf[r \rightarrow w])$; and
- if $(w, u), (u', u') \in G.rf$ and $rng([u'] ; G.porf) = \emptyset$, then $cons(G|_{G.E \setminus \{w\}}.rf[u' \rightarrow w])$.

A model is extensible iff it is read-, write- and rw-extensible.

Finally, the well-blocking condition (MM4) is captured by Def. 3.6. It stipulates that consistent executions satisfy two conditions with respect to blocking reads. First, blocking reads must be maximal in $G.porf$: if an event blocks then it cannot proceed. Second, reads may block only when all writes are matched. That is, if there is a blocking read on $x$ ($G.R_x \not\subseteq dom(G.rf)$), then all writes on $x$ have already read from ($G.w_x \subseteq rng(G.rf)$). Note that when $G.rf$ is a total function, this stipulation is trivially satisfied. As such, this is not a strong requirement: in all well-known memory models as well as the concurrent libraries specified in [38], $G.rf$ is specified to be total.

**Definition 3.6** (Well-blocking). A memory model $m$ is well-blocked iff for all $G$, if $cons_m(G)$ holds, then $G$ is well-blocked.

An execution $G$ is well-blocked iff 1) $[G.B]; G.porf = \emptyset$; and 2) $\forall x \in Loc.G.R_x \not\subseteq dom(G.rf) \lor G.w_x \subseteq rng(G.rf)$. 

Model Checking for Weakly Consistent Libraries PLDI ’19, June 22–26, 2019, Phoenix, AZ, USA
From Programs to Executions Given a concurrent program, we use the same technique as [24] to pre-process it to a program of the form $P = \|_{i \in \mathbb{T}} P_i$, where each $P_i$ is a sequential loop-free deterministic program. The set of executions associated with $P$ is then defined by induction over the structure of sequential programs $P_i$. We omit this formal construction here as it is standard in the literature e.g., [41].

Mutex Library We formulate the notion of mutex library executions and their consistency predicate in Def. 3.7 below. For each mutex at location $l \in \text{Loc}$, the mutex events on $l$ comprise lock and unlock events, where the set of unlock events contains a single initialization event. Given a mutex execution $G = (E, rf)$, we define the mutex consistency predicate such that it holds on $G$ if: 1) $G$ is well-formed (Def. 3.3); 2) $G$ is well-blocked (Def. 3.6); 3) $E$ comprises mutex events; 4) $rf$ is injective; and 5) $rf$ maps lock events on to unlocks.

Intuitively, $rf$ describes the order of mutex acquisition. For each lock event $b$ with $(a, b) \in rf$, if $a$ is an unlock event, then $a$ denotes the event releasing the mutex immediately before it is acquired by $b$: when $a$ is the initialization event, then $b$ corresponds to the very first lock call on the mutex. As such, $rf$ must be an injection.

Note that not all locks may be matched in $rf$. Unmatched locks are blocked, waiting for the mutex release. However, well-formedness ensures that an execution contains blocking locks only when all unlocks are matched (see (2) in Def. 3.6).

Definition 3.7. The mutex event set on $l$ is $\text{MX}_l \triangleq L_l \uplus U_l$ with $L_l \triangleq \{ e \mid \text{lab}(e) = \text{lock}(l) \}$, $U_l \triangleq \{ e \mid \text{lab}(e) = \text{unlock}(l) \}$.

Execution $G$ is mutex-consistent, written $\text{cons}_m(G)$, iff: 1) $G$ is well-formed; 2) $G$ is well-blocked; 3) $G.\text{E} = \bigcup_{l \in \text{Loc}} \text{MX}_l$; 4) $G.rf$ is injective; and 5) $G.rf = \bigcup_{l \in \text{Loc}} rf_l$ for some given $rf_l \subseteq U_l \times L_l$.

It is straightforward to show that $\text{cons}_m(.)$ is well-formed, prefix-closed, extensible and well-blocked.

4 GenMC: The Generic Model Checker

In this section, we present a version of our model checking algorithm, GenMC, that does not record $\text{mo}$. It can be instantiated for any memory model by replacing the consistency checks in the code with MM-specific consistency predicates. We refer the reader to our technical appendix [25] for a version of GenMC that also tracks $\text{mo}$.

Configurations Given a program $P$, recall from §2 that GenMC maintains a configuration comprising an execution $G$ of $P$, and a work list $W$ which stores revisit options both explored or otherwise. As described in §2.3, the options in $W$ are categorized as forward or backward revisits; forward options are removed from $W$ once explored, whilst backwards options are never removed and simply marked as explored.

Formally, we define a configuration as a tuple $(G, T, U, S)$, where $G$ is an execution of $P$; $T$ denotes a set of revisitable reads; $U$ is a map from reads to backward revisits (both explored or otherwise); and $S$ is a map from reads to both forward and backward revisits yet to be explored. As such, when a new revisit candidate is encountered, if it is a forward option, it is added only to $S$, whereas if it is a backward option then it is added to both $S$ and $U$. That is, $S$ serves as a work set (the $W$ map in §2 limited to entries not $\checkmark$-marked).

Analogously, when a revisit is explored, it is only removed from $S$ and not $U$, and thus $U$ retains all backward revisits. For efficiency, the revisitable set $T$ tracks those reads whose incoming $rf$ edges may be changed, i.e., revisit candidates.

Each entry in $S[r]$ (and $U[r]$) is a graph $G'$ representing the effect of revisiting $r$ by a write $w$. As we discuss later in §5, our implementation records only a portion of $G'$ necessary for constructing it from $G$ when $r$ is revisited by $w$. However, for better readability, in our presentation here we record in $G'$ the entire graph resulting from $w$ revisiting $r$.

The next$_p$ Function Recall that we construct graphs by adding events in a fixed order (§2). We define a function, next$_p$, such that given a program $P$ and an execution $G$ of $P$, next$_p(G)$ returns an available event (Def. 3.2) of any thread $i$ in $G$ such that $i$ is not stuck (e.g., due to a failed assume statement) and has not finished execution. When no such thread exists (i.e., all threads are stuck or finished), next$_p$ returns false. We implement next$_p$ to choose the left-most such thread, i.e., one with the smallest thread identifier.

4.1 The Main Verify Procedure

Given a program $P$, we begin exploring the executions of $P$ by calling Verify($P$). This routine creates an initial configuration comprising the $G_0$ graph (containing only the initialization writes), an empty revisit set $T=\emptyset$, and empty maps $U=S=E=\emptyset$ (Line 2). It then generates the executions of $P$ one at a time. This is done by calling VisitOne($P, G, T, U, S$) on Line 3, which fully explores one execution extending $G$, and pushes alternative reads-from-options encountered to the work set $S$. Once VisitOne($P, G, T, U, S$) returns the full execution generated, remaining executions are generated by exploring the options in the work list $S$ Lines 4-10.

Algorithm 1: Main exploration algorithm

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
<td>procedure Verify($P$)</td>
</tr>
<tr>
<td>2:</td>
<td>$(G, T, U, S) \leftarrow (G_0, \emptyset, \emptyset, \emptyset)$</td>
</tr>
<tr>
<td>3:</td>
<td>VisitOne($P, G, T, U, S$)</td>
</tr>
<tr>
<td>4:</td>
<td>while $(r, G') \leftarrow \text{RemoveMax}(S)$ do</td>
</tr>
<tr>
<td>5:</td>
<td>$(E_1, r, E_2) \leftarrow \text{split}(G.E, r)$</td>
</tr>
<tr>
<td>6:</td>
<td>$T \leftarrow T \setminus E_2$</td>
</tr>
<tr>
<td>7:</td>
<td>$U \leftarrow U \setminus {U[r'] \mid r' \in E_2}$</td>
</tr>
<tr>
<td>8:</td>
<td>if $G'.rf[r] \neq \bot$ then</td>
</tr>
<tr>
<td>9:</td>
<td>$\text{CalcRevisits}(G', T, U, S, r)$</td>
</tr>
<tr>
<td>10:</td>
<td>VisitOne($P, G', T, U, S$)</td>
</tr>
</tbody>
</table>
To do this, an option $G'$ is picked from $S[r]$ (Line 4) such that $r$ is the maximal entry in $S$: $r$ is added to the current graph $G$ after all other reads in the domain of $S$. When $S[r]$ holds multiple options, an arbitrary entry is chosen. Picking the maximal entry in $S$ makes it easier to update the current configuration and enables a key optimization (see §5).

We split $G$ at $r$ (Line 5) such that $E_1$ contains events in $G$ added before $r$ and $E_2$ contains those added after $r$. By construction, $E_2$ comprises events that either are not in $G'$ or belong to the porf prefix of the event $a$ that revisited $r$ to generate $G'$. These latter events are responsible for the addition of $a$ to the graph, and consequently the reason why $r$ is revisited. As such, revisiting any of these latter events would “undo” the revisit of $r$. For this reason, we remove the events in $E_2$ from the set $T$ of revisitable reads (Line 6).

Analogously, Line 7 removes the $E_2$ entries from $U$. Note that no such entries exist in $S$: all events in $E_2$ have been added to the graph after $r$, while we picked $r$ to be the maximal entry in $S$; i.e., $S[r']$ contains no entries for $r'$ in $E_2$.

Recall from §2 that when revisiting a (non-blocked) read, we check whether the read being revisited may itself generate backward revisit options for existing reads in the graph. For instance, in the fail/2 and lock/2 examples, revisiting $b$ generated additional revisit options for $a$—see (fail-⊥) and (lock-⊥). This is done by calling CalcRevisits on Line 9.

Finally, on Line 10 we explore the updated configuration.

### 4.2 The VisitOne Procedure

The VisitOne procedure is the workhorse of the exploration algorithm. In each iteration of this loop, while the current graph ($G$) is consistent, it is extended with its next event $a$ (given by next$_P(G)$, see page 8). When next$_P(G)$ returns false, VisitOne terminates. If the next event $a$ is an assertion violation, then an error is reported (Line 3), and the algorithm terminates. Otherwise, we add $a$ to the graph (Line 4). As before, we check whether the newly added event $a$ generates backward revisit options for the existing reads in the graph by calling CalcRevisits on Line 11.

If the new event $a$ is a write, no additional work is required. However, if $a$ is a read, we must calculate its incoming rf edge. We first calculate the set of writes $W$ that a could read from, i.e., its forward revisit options (Line 6), choose a write $w_0$ for the current exploration (Line 7), set $a$ to read from $w_0$ in $G$ (Line 8), add the new read to the revisit set (Line 9), and push the remaining revisits options to $S$ (Line 10).

### 4.3 The CalcRevisits Procedure

As described in §4.1-§4.2, the CalcRevisits routine calculates the set of backward revisits that $a$ can generate and pushes them to $U$ and $S$ unless they have already been considered, i.e., are in $U$ (Lines 7-9).

To calculate the set of revisit graphs, we iterate through all revisitable reads on the same location as $a$ (Line 3), excluding those reads whose revisit would violate porf-irreflexivity; i.e., those in the porf prefix of $a$ calculated in $p_a$ (Line 2). Recall that when $a$ revisits $r$, in the resulting graph we retain $r$, the events added to the graph before $r$ ($E_1$), as well as the events in the porf prefix of $a$ that are added after $r$. To this end, we compute the sets $E_1$ and $E_2$ (Line 4), respectively comprising the events added before and after $r$, and set $G'$ to contain $E_1$, $r$, and the events in both $E_2$ and $p_a$ (Line 5).

If $a$ is a write event, we finally set $r$ to read from $a$ in $G'$ (Line 6). If, however, $a$ is a read, it cannot revisit existing reads in the graph itself but it may cause them to block (cf. lock-⊥), which is why we instead set the incoming rf edge of $r$ to the blocking option ⊥.
given in full in the technical appendix [25]; we proceed with an intuitive argument.

To show that GenMC is complete, we show that it generates all executions of a given program \( P \). As discussed in §2, MM1 and MM2 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 GenMC if we reshuffle the order in which its events are added. That is, for each execution \( G \) generated by adding events in the order \( S = e_1, e_2, \ldots, e_n \), there exists a permutation \( S' \) of \( S \), such that GenMC adds events in the \( S' \) order and generates \( G \). To show that such a reshuffling exists, we often need to remove events from \( G \) and re-add them later (capturing the revisit step). This can always be done thanks to the extensibility property (MM3) ensuring that GenMC never gets stuck.

To show that GenMC is optimal, we observe that duplication can arise only when revisiting a read. As discussed in §2 (see “Avoiding Duplication” on page 4), forward revisits never cause duplication since they are never removed from the graph, while backward revisits may lead to duplication and thus the already-considered backward revisits are recorded in the map \( U \). The optimality of GenMC is thus guaranteed by the properties of forward/backward revisits, the map \( U \) and the check on Line 7 of Algorithm 3.

**Theorem 4.1 (Correctness).** The GenMC algorithm is sound, complete and optimal.

### 5 Implementation

We have implemented GenMC as an open-source verification tool for C programs over the LLVM interpreter 11i. GenMC is available at http://github.com/mpi-sw/genmc.

We have implemented three variants of GenMC:

- **LIB:** a generic variant that performs model checking on libraries, based on specifications provided by the user;
- **WB:** an instantiation for the full RC11 memory model [27], using a consistency check based on \( wb \); and
- **MO:** an alternate RC11 instantiation that records the \( mo \) order during exploration. That is, whenever a write is added to a graph, we consider all its possible placements in \( mo \), and create subexplorations for each case.

Naturally, the generic variant is slower than the RC11 ones because the latter have more optimized consistency checks; it is, however, still optimal.

Further, we have implemented some optimizations over the algorithm described in §4, which we will describe below.

The first key optimization has to do with the representation of the graphs to be revisited in \( S \). In §4, each entry in \( S[r] \) (and \( U[r] \)) is a full graph \( G' \) generated by a forward or backward revisit of the read \( r \). For better space efficiency, rather than recording the entire graph \( G' \), we store only the portion of \( G' \) of events after \( r \), because that suffices for re-constructing the entire \( G' \) when the revisit takes place. The reason is that GenMC revisits executions from \( S \) by always choosing the maximal read in \( S \) when removing an entry from \( S \) (Line 4 of Algorithm 1). The effect of the revisit order is that the current graph projected to the events before \( r \) (i.e., \( E_i \) in Algorithm 1) is exactly the same as the recorded graph \( G' \in S[r] \) projected to the same events. As a result, it suffices to record in each graph in \( S \) only the revisited read and the events after it.

Similarly, we store the graphs in \( U \) in a compressed form. Since we do not ever need to restore the graphs from \( U \), we do not need to store all the events after \( r \); it suffices to record only their incoming \( rf \) edges because those determine the values read and hence the event labels.

Finally, in the WB and MO variants of GenMC, we use optimized consistency checks when adding a new event to the graph. We exploit the fact that the graph prior to adding the event was consistent, so it suffices to check only that the new event does not lead to any consistency violation.

### 6 Evaluation

**Verification Tools**

In the following, we compare the performance of GenMC to three other stateless model checkers: Nidhugg [2], RCMC [24], and Tracer [4]. Initially, we also considered other tools—namely, CBMC [5, 14], CDsChecker [35], and Herd [6]—but exclude them from head-to-head comparisons because they are typically significantly slower than Nidhugg and RCMC and do not scale well (see, e.g., the evaluation in [24]): Herd because it was meant for experimenting only with small “litmus test” programs, CBMC because of the SAT solver, CDsChecker because of its suboptimal partial order reduction technique.

Nidhugg [2] is a state-of-the-art stateless model checker supporting SC, TSO, and PSO.\(^6\) It enumerates \( mo \)-executions (a.k.a. Mazurkiewicz traces [32]) and can operate both under an optimal mode (optimal-DPOR) and a non-optimal mode (source-DPOR). In our benchmarks, we use the source-DPOR version because it is typically faster than the optimal version. Under SC, Nidhugg can also operate under a coarser equivalence partitioning (denoted SC\(^a\) – Nidhugg with observers) [7]. This equivalence can be exponentially coarser than \( mo \)-executions, but remains exponentially finer than plain executions. We used version 0.3 of Nidhugg, and ran it with the \( \neg c11 \) switch, which makes the SC version of Nidhugg noticeably faster.

RCMC [24] targets RC11 and WRC11, a weaker RC11 variant that does not record \( mo \) and does not enforce coherence. RCMC-RC11 also enumerates \( mo \)-executions of a program, though not optimally in the presence of RMW or SC accesses.

Tracer [4] targets RA (the release-acquire fragment of RC11), and enumerates plain executions. It is, however, built over the CDsChecker infrastructure, which makes it quite

\(^6\)Nidhugg also provides some very limited support for POWER, which we do not evaluate because it cannot encode most of our benchmarks.
difficult to apply it fairly to our benchmarks (e.g., it does not support assume statements, and requires manual instrumentation for programs with loops). For this reason, we apply it only to our synthetic benchmarks.

**Benchmarks** We took as benchmarks all the programs from the benchmark suites of Nidhugg and RCMC, together with some additional larger programs (e.g., seqlock, chaselev) from open-source code. In total, we have assembled 127 benchmark programs, some of which are parametric in the number of operations/threads. For suitable values for their parameters, we have generated 202 test cases in total.

First (§6.1), we focus on the generic GenMC variant, and demonstrate how it is used to model check libraries. We conduct a case study for a lock library, and show that abstracting over its implementation has substantial runtime benefits.

Next (§6.2), we evaluate the overall performance of the RC11 variant of GenMC in both synthetic and real-world benchmarks. Our benchmarks highlight the importance of our optimality result, and show that GenMC verifies code currently deployed in production within seconds.

Finally (§6.3), we perform an extensive comparison between the WB and MO variants of GenMC. We show that the WB variant can explore exponentially fewer executions than MO, and the overhead due to its more expensive consistency checks is usually negligible.

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

### 6.1 Model Checking a Lock Library

As a simple demonstration of the benefits of parametricity and compositional verification, we consider a C implementation of Lamport’s fast mutual-exclusion algorithm [29] (see Table 1). We could have considered any correct lock implementation (e.g., the ones used in §6.2), but we chose Lamport’s algorithm because it has write-write races, which are rare in non-synthetic programs and highlight the differences between the various tools. Nidhugg under TSO and PSO are excluded from this table for brevity, as they are slower than Nidhugg-SC.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Nidhugg</th>
<th>RCMC</th>
<th>GenMC</th>
</tr>
</thead>
<tbody>
<tr>
<td>lamport(2)</td>
<td>0.13</td>
<td>0.10</td>
<td>0.03</td>
</tr>
<tr>
<td>lamport(3)</td>
<td>7.53</td>
<td>4.49</td>
<td>6.87</td>
</tr>
<tr>
<td>lamport(4)</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>

### 6.2 Overall Performance

Table 2 reports two synthetic benchmarks, which demonstrate the importance of optimality (Table 2).

In the cinc program, all threads perform a series of RMW operations. Since RCMC is not optimal in the presence of RMWs, it can explore many more executions than necessary, which leads to some runtime overhead. For 4 threads RCMC explores 45% more executions than GenMC, while for 5 threads, it explores almost twice as many executions as GenMC, and this is reflected in the running time. All other tools explore the same number of executions, but Nidhugg is significantly slower than the other tools.

The Nw1r program has \( N + 1 \) concurrent writers and one concurrent reader of a shared variable, and thus has \( (N + 2)! \) mo-executions versus only \( (N+2) \) plain executions. It is therefore not surprising that tools enumerating mo-executions (Nidhugg-SC, RCMC-RC11, and GenMC-MO) do not scale well. Nidhugg-SC explores 193 executions for \( N=5 \) and 2305 for \( N=8 \), and so also does not scale particularly well. In contrast, Tracer, RCMC-WRC11, and GenMC-WB finish almost instantly. Recall, however, that RCMC-WRC11 fails to terminate on other benchmarks (§6.1).

Next, we move to two sets of benchmarks extracted from real programs. Since Nidhugg-SC does not reduce the number of executions and is in fact slower than Nidhugg-SC on these benchmarks, we exclude it from further comparisons.

Table 3 compares the tools on the implementations of concurrent data structures from [13, 35]. We do not show the number of executions explored because all tools explore

### Table 1. Lamport’s fast mutex algorithm [29]

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>SC</th>
<th>SC^0</th>
<th>RC11</th>
<th>WRC11</th>
<th>MO</th>
<th>WB</th>
<th>LIB</th>
</tr>
</thead>
<tbody>
<tr>
<td>lamport(2)</td>
<td>0.13</td>
<td>0.10</td>
<td>0.04</td>
<td>∞</td>
<td>0.03</td>
<td>0.03</td>
<td>0.09</td>
</tr>
<tr>
<td>lamport(3)</td>
<td>7.53</td>
<td>4.49</td>
<td>5.40</td>
<td>∞</td>
<td>6.87</td>
<td>1.36</td>
<td>0.09</td>
</tr>
<tr>
<td>lamport(4)</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>∞</td>
<td>--</td>
<td>--</td>
<td>0.09</td>
</tr>
</tbody>
</table>

### Table 2. Some synthetic benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Nidhugg</th>
<th>Tracer</th>
<th>RCMC</th>
<th>GenMC</th>
</tr>
</thead>
<tbody>
<tr>
<td>cinc(4)</td>
<td>2.98</td>
<td>3.11</td>
<td>1.13</td>
<td>0.69</td>
</tr>
<tr>
<td>cinc(5)</td>
<td>436.40</td>
<td>466.65</td>
<td>165.54</td>
<td>134.87</td>
</tr>
<tr>
<td>Nw1r(5)</td>
<td>1.25</td>
<td>0.17</td>
<td>0.01</td>
<td>0.11</td>
</tr>
<tr>
<td>Nw1r(8)</td>
<td>991.80</td>
<td>0.74</td>
<td>0.01</td>
<td>79.68</td>
</tr>
</tbody>
</table>
the same number of distinct executions, excluding possible redundant executions explored by Nidhugg (under 5%). These benchmarks have the same number of distinct executions regardless of the memory model (i.e., they are robust), which is expected since they only use non-SC accesses for performance reasons. The only exception is chase-lev, for which Nidhugg explores more executions under PSO due to the absence of a store-store fence, which renders the precise modeling of acquire-release operations utilized by this benchmark difficult.

On these benchmarks, RCMC and GenMC outperform Nidhugg, even though they operate under a weaker memory model. By contrast, Nidhugg gets slower as the memory model gets weaker, which is expected due to the way it models TSO and PSO, and agrees with the observations in [2, 24]. GenMC performs similarly in terms of time under WB and MO, and explores the same number of executions. For linuxrwlocks, however, the WB verification requires much more time than MO. This is due to the calculation of \( wb \) as part of RC11’s consistency check, which is particularly slow when there are long chains of RMW events. (In general, calculating \( wb \) can take up to \( O(n^3) \) time in the size of the execution graph, and achieves its worst-case complexity, when there are many writes to the same location.)

Table 4 summarizes the performance of the tools in lock implementations extracted verbatim from the Linux kernel (v4.13.6, v4.19.1). Headers, kernel primitives definitions, macros, and \texttt{Kconfig} options have been provided for all benchmarks as necessary. The test cases involve \( N \) threads accessing shared variables while holding the respective locks.

For all benchmarks, except mcs_spinlock, all tools explore the same number of executions, modulo a few redundant explorations for Nidhugg, and the seqlock test case, where Nidhugg-PSO again explores more executions due to the absence of a store-store fence. As shown, RCMC and GenMC outperform Nidhugg by a large factor.

The mcs_spinlock benchmark is rather interesting for several reasons. First, it allows some relaxed behaviours to take place, and so Nidhugg-PSO, GenMC-MO, and RCMC-MO explore more executions than Nidhugg-SC and Nidhugg-TSO (approximately 15% more). Nonetheless, GenMC and RCMC outperform Nidhugg by a large factor. Second, GenMC-WB and RCMC-WRC11 explore fewer executions than GenMC-MO and RCMC-RC11, and shows the benefit of not recording \texttt{MO} in terms of verification time. Last, GenMC is slower than RCMC on this particular benchmark. This is because GenMC’s revisit procedure removes more events from the graph during backward revisits than RCMC. The extra events must then be re-added resulting in runtime overhead. Of course, this also depends on the nature of the benchmark, and the backward revisits that take place.

### 6.3 Modification Order vs Writes-Before

We next compare GenMC-WB and GenMC-MO more thoroughly. Admittedly, calculating \( wb \) for consistency is much more expensive (\( O(n^3) \)) than using the total order readily given by \texttt{MO}. As we show, however, (a) it can lead to exploring exponentially fewer executions than recording \texttt{MO}; and (b) the overhead imposed by the \( wb \) calculation is usually negligible.

To see (a), consider Fig. 5 (left), depicting the number of executions explored by GenMC-WB and GenMC-MO on some synthetic benchmarks. As shown, for 7 threads, GenMC-MO can visit up to \( 10^6 \) more executions than GenMC-WB, which is also reflected in the running time.
6.5 litmus test programs. Unlike GenMC, we can relax these assumptions to enable verification under well-blocking. In the future, we plan to investigate whether (that do not satisfy \(\text{mm} \)), we can see that GenMC is much faster than GenMC-MO.

The speedup is due to the presence of unordered concurrent writes in the program. Kokologiannakis et al. [24] argue that concurrent writes seldom appear in correct real-world programs, and our benchmarks confirm that claim.

However, there are two observations worth mentioning. First, there are real-world benchmarks (e.g., lamport and \text{mcs\_spinlock}) where there is a difference (although not exponential) in the number of explored executions between GenMC-WB and GenMC-MO, and this difference is reflected in the running time. Second, while correct programs should not have concurrent unordered writes, this may happen in incorrect programs, and observing the difference between the \text{wb} and \text{no} executions can be beneficial to spot such errors.

7 Conclusions and Related Work

We have presented GenMC as an effective model checking approach that is parametric in the choice of memory model and supports high-level concurrent libraries. Our approach relies on four basic assumptions about the underlying memory model: \text{porf}\text{-acyclicity}, extensibility, prefix-closedness, and well-blocking. In the future, we plan to investigate whether we can relax these assumptions to enable verification under hardware memory models such as Power [6] and ARM [37] (that do not satisfy \text{porf}\text{-acyclicity}) and library specifications such as queues [38] (that are not prefix-closed).

Amongst the verification tools handling weak memory models (MMs), the only properly MM-parametric tool is \text{Herd} [6], a memory model simulator that allows users to experiment with different consistency predicates on small \text{“litmus test”} programs. Unlike GenMC, \text{Herd} does not require models to satisfy conditions \text{MM1-MM4}, and so accepts a wider range of models than GenMC. Nevertheless, it follows the simple approach of enumerating all possible executions and filtering them according to the user-supplied consistency predicate, and thus is not scalable when applied to larger programs. It would be worth extending \text{Herd} to use the GenMC approach whenever the user-supplied model can be shown to satisfy conditions \text{MM1-MM4}.

As discussed in \$2, several tools based on \text{stateless model checking} [18, 19, 34] combined with (dynamic) partial order reduction (DPOR) techniques [1, 16] have targeted specific memory models [2–4, 15, 24, 35, 42]. Unfortunately, all of them use somewhat different ideas, making it difficult to get a model checking algorithm that is MM-parametric. Amongst these tools, the only ones enumerating plain executions (as opposed to \text{no}-executions) are: \text{Tracer} [4] for the release-acquire fragment of (R)C11; DC-DPOR [12] for SC; and RCMC [24] for the WRC11 model.

GenMC follows the general design of RCMC, but uses a revisit procedure akin to that of \text{Tracer}, i.e., when in an execution graph \(G\) a write \(w\) revisits a read \(r\), it removes from \(G\) all events that were added to \(G\) after \(r\) and are not \text{porf}\text{-before} \(w\) as opposed to removing only the events \text{porf}\text{-after} \(r\). As a result, the completeness proof of GenMC (unlike that of RCMC) does not require “prefix-determinacy” [24, Lemma 3.9], which does not hold for the entire RC11 model: the weaker “prefix-closedness” suffices. So, while RCMC is optimal only in the absence of RMW and SC accesses, GenMC achieves optimality for the full RC11 model.

Other tools, such as CBMC [14], encode all executions of a program together with the memory model in a \text{SAT/SMT} formula and query a dedicated solver for its satisfiability [5]. This approach should in principle be able to handle models such as RC11; however, it is currently limited to SC, TSO, and PSO. The main drawback of this approach is its \text{SAT/SMT} component, which can be slow and highly unpredictable. As a result, CBMC tends to be significantly slower than \text{Nidhugg} on relevant benchmarks [26, 30].

Another approach is \text{maximal causality reduction} (MCR) [20, 21], which introduces an even coarser equivalence partitioning than \text{porf}, based on \text{values} and not the places reads read-from. This approach fundamentally assumes “multi-copy atomicity” (i.e., that writes propagate simultaneously to all other processors), and thus cannot work for RC11 [27]. It does, however, work well for SC, TSO, and PSO.

Finally, unfolding-based techniques [22, 39] have obtained similar optimality results with some DPOR algorithms for SC. It remains to be seen whether they can be generalized or achieve optimality under a coarser equivalence partitioning.

Acknowledgments

We would like to thank Michael Emmi, Konstantinos Sagonas and the PLDI reviewers for their 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).


A Memory Model Instantiations in Our Framework

A.1 The SC Memory Model

An execution $G$ is SC-consistent iff there exists an order $mo \triangleq \bigcup_{x \in \text{Loc}} mo_x$ such that:

- each $mo_x \subseteq G.W_x \times G.W_x$ is a strict total order on $G.W_x$;
- $G.rf$ is a total function and for all $(w, r) \in G.rf$, $\text{val}_w(w) = \text{val}_r(r)$, where $\text{val}_w(w)$ denotes the value written by $w$ and $\text{val}_r(r)$ denotes the value read by $r$;
- $(G.po \cup G.rf \cup mo \cup (G.rf^{-1}; mo))^+$ is irreflexive.

A.2 The TSO Memory Model

**Definition A.1** (TSO). An execution $G$ is TSO-consistent iff there exists an order $tso \subseteq G.E \times G.E$ such that:

- $tso$ is total on $G.W$;
- $G.po \setminus (G.W \times G.R) \subseteq tso$;
- $G.rf \subseteq tso \cup G.po$;
- $\forall (w, r) \in G.rf \cdot \forall w' \in G.W \cap G.R. (w', r) \in tso \cup G.po \land \text{loc}(w') = \text{loc}(r) \Rightarrow (w, w') \notin tso$;
- $G.rf$ is a total function and for all $(w, r) \in G.rf$, $\text{val}_w(w) = \text{val}_r(r)$, where $\text{val}_w(w)$ denotes the value written by $w$ and $\text{val}_r(r)$ denotes the value read by $r$.

A.3 The RC11 Memory Model

First, let us define the happens-before ($hb$) order for RC11. Intuitively, $hb$ records when an event is globally perceived as occurring before another one. To this end, several derived relations are needed:

$$G.rseq \triangleq \bigcup_{x \in \text{Loc}} [w_x]; G.po^\ast; [w_x^{\text{rlx}}]; (G.rf; [R \cap W])^+ \quad \text{(release sequence)}$$

$$G.sw \triangleq [E^{\text{rel}}]; ([F]; G.po)^\ast; G.rseq; G.rf; [R^{\text{rlx}}]; (G.po; [F])^\ast; [E^{\text{seq}}] \quad \text{(synchronizes with)}$$

$$G hb \triangleq (G.po \cup G.sw)^\ast \quad \text{(happens-before)}$$

Happens-before is defined in terms of two more basic definitions:

1. The release sequence ($rseq$) relation: the release sequence of a write contains the write itself, all later writes to the same location in the same thread, as well as all RMWs that recursively read from such writes.
2. The synchronizes with ($sw$) relation: we say that a release event $a$ synchronizes with an acquire event $b$, whenever $b$ (or, in case $b$ is a fence, some po-prior read) reads from the release sequence of $a$ (or, in case $a$ is a fence, of some po-later write).

Then, we say that an event $a$ happens-before an event $b$ if there is a path from $a$ to $b$ consisting of po and sw edges.

With this definition for $hb$, an RC11-consistent execution must satisfy the requirements described in the following subsections.

A.3.1 Completeness

The first constraint is very simple: every read should read a value from a write with the same value. Accordingly, an execution $G$ is said to be complete if $G.rf$ is a total function and for all $(w, r) \in G.rf$, $\text{val}_w(w) = \text{val}_r(r)$, where $\text{val}_w(w)$ denotes the value written by $w$ and $\text{val}_r(r)$ denotes the value read by $r$.

A.3.2 Coherence

RC11 requires that there exist a total order called modification order ($mo$) that is a disjoint union of relations $\{ mo_x \}_{x \in \text{Loc}}$, such that each $mo_x$ is a strict partial order on $E \cap W_x$.

Then, coherence (a.k.a. SC-per-location) requires two things:

1. For every particular location, all threads agree on the order of accesses to that location, and
2. this order should be consistent with the happens-before order ($hb$).

To order accesses to a given location, the model requires that for every location $x$, $G.mo$ totally orders the writes to $x$, and defines an extension of $mo$, which is a partial order on all accesses to $x$:

$$G.eco \triangleq (G.mo \cup G.rf \cup G.rf^{-1}; G.mo)^+ \quad \text{(extended coherence order)}$$

Here, writes are ordered using $mo$, while reads are placed after the writes they read from, but before writes that are $mo$-later than the writes they read from.

Then, the coherence condition simply requires that $G.eco; G.hb$ is irreflexive.
A.3.3 Atomicity

There must be no two RMW events reading from the same write. Alternatively, there must be no event in modification order between the write from which an RMW \( u \) reads-from and the \( u \) itself.

Formally, we can demand that \( G.\text{rf} ; [R \cap W] ; G.\text{mo}^{-1} ; G.\text{mo}^{-1} \) is irreflexive.

A.3.4 Global SC Constraint

SC accesses and fences are subject to a global constraint, which, roughly speaking, requires threads to agree on their order. In fact, due to the interaction with other access modes, this is notably the most involved part of RC11, which addresses flaws of the original C/C++ memory model.

RC11 requires the acyclicity of a relation called \textit{partial SC order}, denoted \( \text{psc} \), which is, in turn, defined using additional helper notations:

\[
\begin{align*}
G.\text{po}|_{\not= \text{loc}} & \triangleq \{(a,b) \in G.\text{po} \mid \text{loc}(a) \neq \text{loc}(b)\} \\
G.\text{hb}|_{\text{loc}} & \triangleq \{(a,b) \in G.\text{hb} \mid \text{loc}(a) = \text{loc}(b)\} \\
G.\text{scb} & \triangleq G.\text{po} \cup G.\text{po}|_{\not= \text{loc}}; G.\text{hb}; G.\text{po}|_{\not= \text{loc}} \cup G.\text{hb}|_{\text{loc}} \cup G.\text{mo} \cup G.\text{rf}^{-1}; G.\text{mo} \quad \text{(SC-before)} \\
G.\text{psc} & \triangleq ([E^{\text{sc}}] \cup [F^{\text{sc}}]; G.\text{hb}^7); G.\text{scb}; ([E^{\text{sc}}] \cup G.\text{hb}^7; [F^{\text{sc}}]) \cup [F^{\text{sc}}]; (G.\text{hb} \cup G.\text{hb}; G.\text{eco}; G.\text{hb}); [F^{\text{sc}}] \quad \text{(partial SC order)}
\end{align*}
\]

A.3.5 No \textit{porf} cycles

In order to rule out "out-of-thin-air" behaviours, RC11 requires that \textit{porf} be acyclic.
B The Writes-Before Relation

Given the RC11 memory model in §A.3, the writes-Before (wb) relation is an alternative coherence definition that ensures the existence of modification orders, without actually providing them. For a memory location $x$, $wb_x$ can be defined as follows:

$$wb_x \triangleq [G.W_x]; \left( (G.rf^*; G.hb; (G.rf^{-1})^2; G.rf^*) \setminus (G.rf^{-1})^* \right); [G.W_x]$$
C Tracking the Modification Order

1: procedure Verify(P)
2:   (G, T, U, S) ← (G₀, ∅, ∅, ∅)
3:   VisitOne(P, G, T, U, S)
4:   while el ← RemoveMax(S) do
5:     if el = ⟨r, w, Gₜ⟩ then
6:       ⟨E₁, r, E₂⟩ ← split(G, E, r)
8:       G, rf[r] ← w
9:       T ← T ∩ (E₁ + [r])
10:      U ← U \ {U[e'] | e' ∈ E₂}
11:     if w ≠ ⊥ then CalcRevisits(G, T, U, S, r)
12:   if el = ⟨w, M⟩ then
13:      ⟨E₁, w, E₂⟩ ← split(G, E, w)
14:      G ← E₁ + [r], G, rf[E₁], G, mo[E₁]
15:      G, mo[loc(w)] ← M × {w} ∪ {w} × (E₁ ∩ W₁loc(w) \ M)
16:      T ← T ∩ (E₁ + [w])
17:      U ← U \ {U[e'] | e' ∈ E₂}
18:      CalcRevisits(G, T, U, S, w)
19:   VisitOne(P, G, T, U, S)

1: procedure VisitOne(G, T, U, S)
2:   while cons(G) ∧ a ← nextₚ(G) do
3:     if a ∈ error then exit(“erroneous program”)
4:     G ← Add(G, a)
5:     if a ∈ R then
6:       W ← G, E ∩ W₁loc(a)
7:       B ← {⟨⊥, ∅⟩ | cons(G, rf[a ↦ ⊥])}
8:       choose some w₀ ∈ W
9:       G, rf[r] ← w₀
10:      T ← T ∪ {r}
11:      S[a] ← S[a] ∪ {⟨w, ∅⟩ | w ∈ W \ {w₀}} ∪ B
12:      U[a] ← U[a] ∪ B
13:     if a ∈ W then
14:       G, mo ← G, mo ∪ (G, E ∩ W₁loc(a) × {a})
15:       S[a] ← S[a] ∪ {M | M ⊆ G, E ∩ W₁loc(a) ∧ rng([M]; G, mo) ⊆ M}
16:      CalcRevisits(G, T, U, S, a)

1: procedure CalcRevisits(G, T, U, S, a)
2:   p_a ← dom(G, porf²[a])
3:   for r ∈ T ∩ R₁loc(a) \ p_a do
4:     ⟨E₁, r, E₂⟩ ← split(G, E, r);
5:     G_a ← G | E₁ ∩ p_a
6:     if a ∉ W then a ← ⊥
7:     if ⟨a, G_a⟩ ∉ U[r] ∧ ⟨a, ∅⟩ ∉ U[r] then
8:       S[r] ← S[r] ∪ {⟨a, G_a⟩}
9:       U[r] ← U[r] ∪ {⟨a, G_a⟩}
D Completeness and Optimality Proofs

In what follows, we write \( G, \text{pred} \) for the strict total order induced by the \( G, E \) sequence. That is, \( (a, b) \in G, \text{pred} \iff a <_{G, E} b \), where \( a <_{G, E} b \) denotes that \( a \) appears before \( b \) in \( G, E \).

D.1 Completeness Proof

**Proposition D.1** (Read-extensibility). Given execution \( G \) such that \( \text{cons}(G) \) holds, and event \( r = (i, |G, E|_i + 1, l) \) with \( \text{typ}(r) = \text{R} \), there exists \( w \in G, \text{loc}(r) \) such that \( \text{cons}(\text{Add}(G, r), \text{rf}[r \mapsto w]) \) holds.

**Proposition D.2** (Write-extensibility). Given execution \( G \) and event \( w \), with \( \text{typ}(w) = \text{W} \) and \( \text{rng}([w]; \text{porf}) = \emptyset \), if \( \text{cons}(G \setminus \{w\}) \) holds, then so does \( \text{cons}(G) \).

**Proposition D.3** (Update-extensibility). Given execution \( G \) such that \( \text{cons}(G) \) holds, and event \( u \) with \( \text{typ}(u) = \text{RMW} \) and \( \text{rng}([u]; G, \text{po}) = \emptyset \), then the following also hold:

(a) Given event \( r \) such that:
- \( \text{typ}(r) = \text{R} \)
- \( (u, r) \in G, \text{rf} \)
- \( \text{rng}([r]; G, \text{po}) = \emptyset \)

then \( \exists w \in \text{G,E} \setminus \{u\} \) such that \( \text{cons}(G, \text{rf}[r \mapsto w]) \).

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

then \( \text{cons}(G \setminus \{u\}, \text{rf}[u' \mapsto w]) \).

**Definition D.4** (Prefix construction). Given an execution \( G_f \supseteq G_0 \), an event \( e \) is added to \( G_0 \) in \( \text{porf}\text{-order} \), written \( G_0 \xrightarrow{e} G_1 \), if \( e \in G_f \setminus G_0 \), \( \text{dom}(G_f, \text{porf}; [e]) \subseteq G_0 \), and \( G_1 \setminus \{e\} = G_0 \).

A graph \( G \) is constructed in \( \text{porf}\text{-order} \) if \( \emptyset \xrightarrow{} G \), where \( \xrightarrow{} \) denotes the reflexive-transitive closure of \( \xrightarrow{} \) (i.e., \( G \) is constructed via a succession of \( \text{porf}\text{-steps} \)).

**Definition D.5** (Algorithmic construction). Given an execution \( G_0 \), an event \( e \) is added to \( G_0 \) in \( \text{algorithmic order} \), written \( G_0 \xrightarrow{e} G_1 \), if there exists \( T, S, U, T', S', U' \) such that Algorithm 2 calls \( \text{VisitRead} \) (resp. \( \text{VisitWrite} \) ) from a configuration \( (G_0, T, S, U) \), and a configuration \( (G_1, T', S', U') \) is produced either immediately after the call, or when \( \text{RevisitReads} \) pops one of the entries pushed to \( S \) by \( \text{VisitRead} \) (resp. \( \text{VisitWrite} \) ). Note that, in either case, it will be \( G_1 \setminus \{e\} = G_0 \).

Given a graph \( G_0 \), an event \( e \) is added in algorithmic order with \( \text{no revisits} \), written \( G_0 \xrightarrow{e, \text{nr}} G_1 \), if \( G_0 \xrightarrow{e} G_1 \) without revisiting any events along the way.

A graph \( G \) is constructed in \( \text{algorithmic order} \) if \( \emptyset \xrightarrow{} G \), where \( \xrightarrow{} \) denotes the reflexive-transitive closure of \( \xrightarrow{} \) (i.e., there is a run of Algorithm 2 that produces a configuration \( (G, \cdot, \cdot, \cdot) \)).

Lastly, given an algorithmic construction \( G_0 \xrightarrow{} G_n \), a graph \( G_1 \) is part of this construction, written \( G_1 \in [G_0 \xrightarrow{} G_n] \), if it is \( G_0 \xrightarrow{e} G_1 \Rightarrow^+ G_n \).

**Theorem D.6** (Completeness). For any \( G_f \in \llbracket P \rrbracket \) such that \( \text{cons}(G_f) \) it is \( \emptyset \xrightarrow{} G_f \).

**Assume:** 1. \( P \) is a deterministic program with finite traces
   2. \( G_f \in \llbracket P \rrbracket \)
   3. \( \text{cons}(G_f) \)

**Prove:** \( \emptyset \xrightarrow{} G_f \)

(1.1) \( \emptyset \xrightarrow{} G_f \)

**Proof:** Can be shown trivially by induction on the length of \( G_f \).

(1.2) Q.E.D.

**Proof:** From Lemma D.7, by taking \( G = G_f = G_{\text{ext}} \).

\( \Box \)

**Lemma D.7** (Prefix extension). If \( \emptyset \xrightarrow{} G \Rightarrow^+ G_{\text{ext}} \) with \( \text{cons}(G) \) and \( G_{\text{ext}} \in \llbracket P \rrbracket \) for some program \( P \), then \( \emptyset \xrightarrow{} G_{\text{ext}} \).

**Proof sketch:** Proof by induction: we assume that we have a fixed prefix construction \( G \) from which the algorithm can reach \( G_{\text{ext}} \), 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 $\emptyset \Rightarrow^* G \xrightarrow{c} G'$, and a graph $G'^\text{ext}$ such that $G' \Rightarrow^* G'^\text{ext}$, we show that it is also $G \Rightarrow^* G'^\text{ext}$, and then use the inductive hypothesis to prove that $\emptyset \Rightarrow^* G'^\text{ext}$.

**Assume:** 1. $P$ is a deterministic program with finite traces
2. $\emptyset \Rightarrow^* G$ with $\text{cons}(G)$
3. $G \Rightarrow^* G'^\text{ext}$ with $G'^\text{ext} \in [\llbracket P \rrbracket]$

**Prove:** $\emptyset \Rightarrow^* G'^\text{ext}$

(1.1) **Case:** $G = \emptyset$

**Proof:** Trivially holds from assumption 3 and case assumption (1).

(1.2) **Case:** $\emptyset \Rightarrow^* G \xrightarrow{c} G'$

2. **IH:** For any $G'^\text{ext}$ such that $G \Rightarrow^* G'^\text{ext}$, it is $\emptyset \Rightarrow^* G'^\text{ext}$

**Proof:** From D.8 by taking $G_0 = G$, $G_f = G'^\text{ext}$, $G = \emptyset$ and $G' = G'$ we get that $G \Rightarrow^* G_f$. Then, we get the desired result from the IH.

(1.3) **Q.E.D.**

**Proof:** Steps (1.1), (1.2), assumptions 1-3, and mathematical induction.

\[ \square \]

**Proposition D.8 (Commutativity).** If $G_0 \Rightarrow^* G \xrightarrow{c} G' \Rightarrow^* G_f$ for some $G_0, G_f$, with $\text{cons}(G_0), \text{cons}(G_f)$, and $G_f \in [\llbracket P \rrbracket]$, 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 $e$, 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 $G'$.

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

**Prove:** $G_0 \Rightarrow^* G_f$

(1.1) **Case:** $G' = G_f$

**Proof:** Algorithm 2 can simply take the step; it is $G \xrightarrow{c} G'$. From case hypothesis (1) and assumption (0):4 we get that $G_0 \Rightarrow^* G_f$.

(1.2) **Case:** $G' \xrightarrow{a} G_2 \Rightarrow^* G_f$

2. **IH:** For any $G_3$ such that $G_0 \Rightarrow^* G_3 \xrightarrow{c} G'_3 \Rightarrow^* G_f$, it is $G_0 \Rightarrow^* G_f$

(2.1) **Case:** next$_P(G) = e$

**Proof:** It is $G \Rightarrow G'$ from graph extensibility. Thus, from (1):1 we get that $G_0 \Rightarrow^* G \xrightarrow{c} G' \xrightarrow{a} G_2 \Rightarrow^* G_f$.

(2.2) **Case:** next$_P(G) = a$ with $a \neq e$

(3.1) **Case:** $\langle e, a \rangle \not\in G_2.r.f$ and $a$ performs no revisiting

(4.1) There exists $G''$ such that $G \xrightarrow{a} G'' \xrightarrow{c} G_2$

**Proof:** Graph extensibility guarantees that $a$ can be added in a consistent manner in $G'$ and, in addition, in the case where $\text{typ}(a) \in \{R, RW\}$, Def. 3.4 with assumption (3) guarantees that $\exists w \in G''$, $G_2$ such that $\langle w, a \rangle \in G'' . r.f$ and $\langle w, a \rangle \in G_2.r.f$ (i.e., $G'' = G_2 \setminus \{e\}$). From case assumptions (1):1 and (3) we get that it is $G'' \xrightarrow{c} G_2$, i.e., the two transitions commute: both $G \xrightarrow{a} G'' \xrightarrow{c} G_2$ and $G \xrightarrow{c} G' \xrightarrow{a} G_2$ hold.

(4.2) **Q.E.D.**

**Proof:** From (4.1) and IH (assumption (1):2).

(3.2) **Case:** $\langle e, a \rangle \in G_2.r.f$

(4.1) There is a run of Algorithm 2 such that $G \xrightarrow{\gamma} G'' \Rightarrow_{nr}^* G_1 \xrightarrow{c} G_2$

**Proof:** From Prop. D.9, 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 $\langle e, a \rangle \not\in G_2.r.f$
Algorithm 3 will not discard \(a\)

**Proof:** From graph extensibility (Prop. D.2). (Intuitively, this is the graph in which Algorithm 3 will revisit all consistent reads.)

(4.2). \(e \in \text{rng}([r_0]; G'_2, \text{pred})\)

From assumptions (0):3 and (1):1.

(4.3). **Assume:** \(e \in \text{dom}(G'_2, \text{porf}; \{a\})\)

**Proof:** False

**Proof:** From case assumption (3) it is \(\text{typ}(a) = W\). Then, case assumption (4) implies that it must be \(a \in \text{rng}([e]; G'_2, \text{porf}; G_2')\). However, this in turn means that \(\text{rng}([e]; G'_2, \text{porf}) \neq \emptyset\), which contradicts assumption (1):1.

(4.4). \(e \notin G_2\)

**Proof:** When a revisit takes place for all \(e'\) such that \(e' \in \text{rng}([r_0]; G'_2, \text{pred})\) and \(e' \notin \text{dom}(G'_2, \text{porf}; \{a\})\) it is \(e' \notin G_2\) (Algorithm 3 and §4.3). Since for \(e\) it also is \(e \in \text{rng}([r_0]; G'_2, \text{pred})\) and \(e \notin \text{dom}(G'_2, \text{porf}; \{a\})\) (steps (4.2) and (4.3)), it is \(e \notin G_2\).

(5). Q.E.D.

**Proof:** From step (4.4) we get that it is \(G' \not\Rightarrow G_2\). 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 \(G' \not\Rightarrow G'' \not\Rightarrow G_2\)

**Proof:** Note that, since \(a\) revisits \(e\), it must be \(\text{typ}(a) \in \{W, RR\}\). However, from prefix construction (assumption (0):3) we get that \(\text{typ}(a) = W\). Thus, from Prop. D.2, we get that \(G' \not\Rightarrow G''\). Then, from case assumptions (1):1 and (3), we get that the two transitions commute, i.e., both \(G' \not\Rightarrow G'' \not\Rightarrow G'\) and \(G' \not\Rightarrow G_2\) 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.

(2.3). Q.E.D.

**Proof:** Cases (2.1) and (2.2) are exhaustive.

(1.3). Q.E.D.

**Proof:** Steps (1.1), (1.2), and mathematical induction.

\[\square\]

**Proposition D.9** (Revisitable extension). If \(\not\Rightarrow^* G \overrightarrow{G' \Rightarrow G_2 \Rightarrow G_f}, \text{ with } a <_G e \text{ and } (e, a) \in G_2.\text{rf}, \text{ then there is a run of Algorithm 2 from } G \text{ such that } G \not\Rightarrow G'' \not\Rightarrow \not\Rightarrow^*_G G_1 \not\Rightarrow G_2\).

**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 3 will not discard \(a\) from the set of revisitable options.

**Assume:** 1. \(P\) is a deterministic program with finite traces

2. \(\not\Rightarrow^* G \overrightarrow{G' \Rightarrow G_2 \Rightarrow G_f}\)

3. \(a <_G e\)

4. \((e, a) \in G_2.\text{rf}\)

**Proof:** There exists a run of Algorithm 2 from \(G\) such that \(G \not\Rightarrow G'' \not\Rightarrow \not\Rightarrow^*_G G_1 \not\Rightarrow G_2\)

(1.1). It is next\(_P\)(\(G\)) = \(a\)

**Proof:** From assumptions (0):2 and (0):3.

(1.2). There is at least one run \(G \not\Rightarrow G'' \not\Rightarrow \not\Rightarrow^*_G G_1 \text{ with next}_P(G'_1) = e\)

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

(1.3). **Case:** \(\text{typ}(e) = W\)

2.1. Choose graphs \(G'_1\) and \(G_p\) such that

1. \(G \not\Rightarrow G_1 \Rightarrow \not\Rightarrow^*_G G'_1\)
2. $G_i \xrightarrow{\text{nr}} G_p$

   Proof: Step (1)2 and graph extensibility (Prop. D.2) guarantee that we can select appropriate $G_i'$ and $G_p$.

(2.2). It is $a \in T$ when $G_i'$ is reached in the production $G \xrightarrow{a} G_1 \xrightarrow{\text{nr}} G_i'$

   Proof: From step (2.2)1 we know that no revisiting is performed, and only if an event gets revisited it is removed from $T$.

(2.3). It is $\text{cons}(G_2)$

   Proof: By assumption (0):2.

(2.4). Assume: $(a, e) \in G_p$porf

   Proof: False

   (3.1). It is $e \in \text{rng}([a]; G_p$porf$); G_p$porf$)

   Proof: This implies that revisiting is performed, which contradicts case assumption (2):1.

   (3.2). It is $e \in \text{rng}([a]; G_p$porf$); G_p$rf$)

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

   (3.3). Q.E.D.

   Proof: From assumption (2) we get that cases (3)1 and (3)2 are exhaustive.

(2.5). Q.E.D.

   It will be $G_i' \xrightarrow{\text{nr}} G_2$, because $a$ will be considered as an event to revisit by Algorithm 3, and will not be discarded (steps (2):2, (2):3, (2):4).

(1.4). Case: $\text{typ}(e) = \text{RMW}$

   (2.1). Let $w \in G.E$ such that $(w, e) \in G'$rf$

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

(2.2). There exists $m \in \mathbb{N}$ and $G_0, G_3, G_4$ such that

   1. $G_0 \xrightarrow{\text{nr}} G_3$

   2. $\text{tid}(\text{next}_G(G_3)) = \text{tid}(e)$

   3. One of the following holds

      a. $G_0 = G'$ and $\exists r \in G_3$rf such that $(e, r) \in G_3$rf

      b. $G_0 = G' \setminus \{e\}$ and $e \notin G_3$E and $\exists u' \in G_3$E such that $(w, u') \in G_3$rf

   Proof: From assumption (0):2, case assumption (1), step (2):1, and by choosing an appropriate $n$ (which is always possible due to program finiteness), Prop. D.10 yields the required result.

(2.3). Choose $G_i'$ such that $G_i' = G_3 \setminus \{e\}$, $G \xrightarrow{\text{nr}} G_i'$, and one of the following holds:

   1. $\exists r \in G_i'$E such that $(e, r) \in G_i'$rf$

   2. $\exists u' \in G_i'$E such that $(w, u') \in G_i'$rf$

(3.1). Choose $G_i'$ such that $G_i' = G_3 \setminus \{e\}$

   Proof: From Def. 3.4 and (2.2):3.

(3.2). Case: $G_0 = G'$ and $\exists r \in G_3$E such that $(e, r) \in G_3$rf

   (4.1). It is $G_0 \setminus \{e\} \xrightarrow{\text{nr}} G_i'$ and $\exists r \in G_i'$E such that $(e, r) \in G_i'$rf

   Proof: From step (2):2.1 and case assumption (3) we know that we can use Lemma D.11.

   (4.2). Q.E.D.

   Proof: From step (4):1, along with assumption (0):2 and case assumption (3).

(3.3). Case: $G_0 = G' \setminus \{e\}$ and $e \notin G_3$E and $\exists u' \in G_3$E such that $(w, u') \in G_3$rf

   Proof: By choosing $G_i' = G_3$, since $e \notin G_3$ and no revisiting is performed (assumptions (3) and (2):2.1), and also $G = G_0$ (assumptions (0):2 and (3)), it trivially is $G \xrightarrow{\text{nr}} G_i'$.

   (3.4). Q.E.D.

   Proof: From (2):2.3 we get that cases (3):2 and (3):3 are exhaustive.

(2.4). There is at least one $G_p$ such that $G_i' \xrightarrow{\text{nr}} G_p$


(2.5). It is $a \in T$ when $G_i'$ is reached in the production $G \xrightarrow{a} G_1 \xrightarrow{\text{nr}} G_i'$

   Proof: From (2):3 and (1):1. Note that, if there are more than one alternatives for the first step of the production $G \xrightarrow{a} G_1 \xrightarrow{\text{nr}} G_i'$, it will be $a \in T$ in one of them.

(2.6). It is $\text{cons}(G_2)$

   Proof: From assumption (0):2.

(2.7). We can choose $G_p$ such that either $(a, e) \notin G_p$porf or $w \in W_e$ when $e$ is added

   (3.1). Case: $\exists r \in G_i'$rf such that $(e, r) \in G_i'$rf
(4.1) There exists $G_p$ such that $\langle w, e \rangle \in G_p.\text{rf}$
Proof: From steps (2.2) and (2.4), as well as case assumption (3), we can choose $G_p = G_3$.

(4.2) Q.E.D.

Proof: From case assumption (3), and steps (2.4) and (4.1), it cannot be $\langle a, e \rangle \in G_p.\text{porf}$.

(3.2) Case: $e \notin G_1.\text{E}$ and $\exists u' \in G_1.\text{E}$ such that $\langle w, u' \rangle \in G_1.\text{rf}$

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

(3.3) Q.E.D.

Proof: From (2.3) we get that cases (3.1) and (3.2) are exhaustive.

(2.8) Q.E.D.

(1.5) Q.E.D.

Proof: From (0):4 we conclude that cases (1)3 and (1)4 are exhaustive.

□

Proposition D.10 (Non-sbrf extension). If $\emptyset \xrightarrow{*} G \xrightarrow{\xi} G_1$, with $\text{typ}(e) = \text{RMW}$ and $\langle w, e \rangle \in G_1.\text{rf}$ for some $w \in G_1.\text{E}$, 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 $a \in E$ such that $G_2 \xrightarrow{a} nr G_3 \land \text{tid}(a) = \text{tid}(e)$
3. One of the following holds
   a. $G_0 = G_1$ and $\exists r \in G_2.\text{E}$ such that $\langle e, r \rangle \in G_2.\text{rf}$
   b. $G_0 = G_1 \setminus \{e\}$ and $e \notin G_2.\text{E}$ and $\exists u' \in G_2.\text{E}$ such that $\langle w, u' \rangle \in G_2.\text{rf}$

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. $\emptyset \xrightarrow{*} G \xrightarrow{\xi} G_1$
3. $\text{typ}(e) = \text{RMW}$
4. $\langle w, e \rangle \in G_1.\text{rf}$ for some $w \in G_1.\text{E}$
5. There exists $n \in \mathbb{N}$ such that $G_1 \Rightarrow nr^n G'_1$

Proof: 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 $a' \in E$ such that $G_2 \xrightarrow{a'} nr G_3 \land \text{tid}(a') = \text{tid}(e)$
3. One of the following holds
   a. $G'_0 = G_1$ and $\exists r' \in G'_2.\text{E}$ such that $\langle e, r' \rangle \in G'_2.\text{rf}$
   b. $G'_0 = G_1 \setminus \{e\}$ and $e \notin G'_2.\text{E}$ and $\exists u' \in G'_2.\text{E}$ such that $\langle w, u' \rangle \in G'_2.\text{rf}$

(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 $\exists r \in G_1.\text{E}$ such that $\langle e, r \rangle \in G_1.\text{rf}$ (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 $a' \in E$ such that $G'_2 \xrightarrow{a'} nr G'_3 \land \text{tid}(a') = \text{tid}(e)$
   c. One of the following holds
      i. $G'_0 = G_1$ and $\exists r' \in G'_2.\text{E}$ such that $\langle e, r' \rangle \in G'_2.\text{rf}$
      ii. $G'_0 = G_1 \setminus \{e\}$ and $e \notin G'_2.\text{E}$ and $\exists u' \in G'_2.\text{E}$ such that $\langle w, u' \rangle \in G'_2.\text{rf}$

(2.1) Case: $m' = k$

(3.1) Choose $G'_3$ such that $G'_2 \xrightarrow{e'} nr G'_3$

Proof: From assumptions (0):5, (1.1), and graph extensibility we get that Algorithm 2 can indeed take a step.
(3.2). Case: tid(e') = 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 \Rightarrow_{\text{nr}} G'_2 \land tid(a'') = tid(e') (assumptions (1):2c)
2. One of the following holds (from assumption (1):2c)
   a. G''_0 = G_1 \land \exists r \in G''_2.\mathcal{E} such that (e, r) \in G''_2.\mathcal{RF}
   b. G''_0 = G_1 \setminus \{e\} and e \notin G''_2.\mathcal{E} and \exists u' \in G''_2.\mathcal{E} such that (w, u') \in G''_2.\mathcal{RF}

(3.3). Case: tid(e') \neq tid(e)

(4.1). Case: G''_0 = G_1 \land \exists r \in G''_2.\mathcal{E} such that (e, r) \in G''_2.\mathcal{RF}
Proof: By choosing m'' = k + 1, G''_0 = G'_0, G''_2 = G'_2, G''_3 = G'_3 we get

1. G''_0 \Rightarrow_{\text{nr}} m'' G''_2 (assumptions (1):2a, (3), and step (3)1)
2. m'' = n (assumption (1):1)
3. G''_0 = G_1 \land \exists r \in G''_2.\mathcal{E} such that (e, r) \in G''_2.\mathcal{RF} (from step (3)1, case assumption (5) and Prop. D.3–Item (a), as well as case assumptions (3) and (4))

(4.2). Case: typ(e') = R
Proof: By choosing m'' = k + 1, G''_0 = G'_0, G''_2 = G'_2, G''_3 = G'_3 we get

1. G''_0 \Rightarrow_{\text{nr}} m'' G''_2 (assumptions (1):2a, (3), and step (3)1)
2. m'' = n (assumption (1):1)
3. G''_0 = G_1 \land \exists r \in G''_2.\mathcal{E} such that (e, r) \in G''_2.\mathcal{RF} (from steps (3)1 and (6)2, as well as case assumptions (3) and (4))

(4.3). Case: typ(e') = RMW

(6.1). Exists w' \in G'_2.\mathcal{E} such that (w', e') \in G'_2.\mathcal{RF}
Proof: From (3)1, case assumption (5), and graph extensibility.

(6.2). Case: w' \neq e
Proof: By choosing m'' = k + 1, G''_0 = G'_0, G''_2 = G'_2, G''_3 = G'_3 we get

1. G''_0 \Rightarrow_{\text{nr}} m'' G''_2 (assumptions (1):2a, (3), and step (3)1)
2. m'' = n (assumption (1):1)
3. G''_0 = G_1 \land \exists r \in G''_2.\mathcal{E} such that (e, r) \in G''_2.\mathcal{RF} (from steps (3)1 and (6)2, as well as case assumptions (3) and (4))

(7.1). Let G''_0 \triangleq G'_0 \setminus \{e\}.\mathcal{RF}[e' \mapsto w] such that G''_0 \Rightarrow_{\text{nr}} m' G''_2 and \exists u' \in G''_2.\mathcal{E} such that (w, u') \in G''_2.\mathcal{RF}
Proof: From (1):2a we know that G'_0 \Rightarrow_{\text{nr}} k G'_2. Then, from graph extensibility (Prop. D.3–Item (b)) with case assumptions (6), (5), (3), and assumptions (6):3 and (6):4 we get the desired result, with u' = e'.

(7.2). Q.E.D.
Proof: By choosing m'' = k + 1, G''_0 = G'_0 \setminus \{e\} and G''_2 from (7)1 we get

1. G''_0 \Rightarrow_{\text{nr}} m'' G''_2 (from Lemma D.11 with step (7)1, and case assumptions (4) and (1):2a)
2. m'' = n (assumption (1):1)
3. G''_0 = G_1 \setminus \{e\} and e \notin G''_2.\mathcal{E} and \exists u' \in G''_2.\mathcal{E} such that (w, u') \in G''_2.\mathcal{RF} (from case assumption (4) and step (7)1)

(7.4). Q.E.D.
Proof: Cases (6)2 and (6)3 are exhaustive.

(5.4). Q.E.D.
Proof: Cases (5)1–(5)3 are exhaustive.

(4.2). Case: G''_0 = G_1 \setminus \{e\} and e \notin G''_2.\mathcal{E} and \exists u' \in G''_2.\mathcal{E} such that (w, u') \in G''_2.\mathcal{RF}
Proof: By choosing m'' = k + 1, G''_0 = G'_0, G''_2 = G'_2 we get

1. G''_0 \Rightarrow_{\text{nr}} m'' G''_2 (case assumptions (1):2a, (3), and step (3)1)
2. m'' = n (assumption (1):1)
3. G''_0 = G_1 \setminus \{e\} and e \notin G''_2.\mathcal{E} and \exists u' \in G''_2.\mathcal{E} such that (w, u') \in G''_2.\mathcal{RF} (from graph extensibility, step (3)1, and case assumptions (3) and (4))

(4.3). Q.E.D.
Proof: From (1):2c we get that cases (4)1 and (4)2 are exhaustive.

(3.4). Q.E.D.
Proof: Cases (3)2 and (3)3 are exhaustive.
Lemma D.11 (Equivalent production). If there are \( n \in \mathbb{N}, G_0, G_n, e \in E \) such that \( \text{cons}(G_0), G_0 \Rightarrow_{nr} G_n \) and for all \( G_i \in [G_0 \Rightarrow_{nr} G_n] \), it is \( \text{next}_p(G_i) \neq e \) and either \( \text{rng}([e]; G_i; \text{porf}) = \emptyset \) or \( e \notin G_i.E \), then there is an equivalent production \( G_0 \setminus \{e\} \Rightarrow_{nr} G_n \setminus \{e\} \).

**Proof sketch:** Simple proof by induction.

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

**Prove:**
1. Case: \( n = 0 \)

   **Proof:** It obviously is \( G_0 = G_n \) (assumption 3). From prefix-closedness (Def. 3.4) and assumption 2, it also is \( \text{cons}(G_0 \setminus \{e\}) \).
2. Case: \( n = k + 1 \)
   1. IH: Given a \( G' \) such that \( G_0 \Rightarrow_{nr} G' \), where for all \( G_i' \in [G_0 \Rightarrow_{nr} G'] \) it is
      a. \( \text{next}_p(G_i') \neq e \)
      b. \( \text{rng}([e]; G_i' ; \text{porf}) = \emptyset \) or \( e \notin G_i'.E \)

   **Proof:** From assumptions (0):3 and (1):1.
3. 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 \( G_0 \setminus \{e\} \Rightarrow_{nr} G_k \setminus \{e\} \) (assumption (1):2) with \( \text{cons}(G_k \setminus \{e\}) \) (courtesy of Algorithm 2), it also is \( G_0 \setminus \{e\} \Rightarrow_{nr} G_k \setminus \{e\} \setminus \{e\} \) (from step (2):1).
4. Case: \( \text{rng}([e]; G_{k+1}; \text{porf}) = \emptyset \)
   **Proof:** From (2):1.
5. **Proof:** From prefix-closedness (Def. 3.4) we get that it is \( \text{cons}(G_{k+1} \setminus \{e\}) \), and from the IH (assumption (1):2), we get that \( G_0 \setminus \{e\} \Rightarrow_{nr} G_k \setminus \{e\} \) with \( \text{cons}(G_k \setminus \{k\}) \) (algorithmic construction). Thus, from step (3):1, it can also be \( G_0 \setminus \{e\} \Rightarrow_{nr} G_{k+1} \setminus \{e\} \).

   **Q.E.D.**

**Proof:** From assumption (0):3b and step (2):1 we get that cases (2):2 and (2):3 are exhaustive.

(1) **Proof:** Steps (1):1, (1):2 and mathematical induction.
Algorithm 4 Explore one program execution

1: procedure VisitOne($G, T, U, S, A$)
2:     while cons($G$) $\land$ $a \leftarrow$ next$_p(G$) do
3:         if $a \in \text{error}$ then exit ("erroneous program")
4:             $G \leftarrow$ Add($G, a$)
5:         if $a \in \mathbb{R}$ then
6:             $W \leftarrow G.E \cap W_{loc}(a)$
7:             $B \leftarrow \{\langle \bot, \emptyset \rangle \mid \text{cons}(G.\text{rf}[a \mapsto \bot])\}$
8:             choose some $w_0 \in W$
9:             $G.\text{rf}[r] \leftarrow w_0$
10:            $T \leftarrow T \cup \{r\}$
11:            $S[a] \leftarrow S[a] \cup \{(w, \emptyset) \mid w \in W \setminus \{w_0\}\} \cup B$
12:            $U[a] \leftarrow U[a] \cup B$
13:            CalcRevisits($G, T, U, S, A$)
14:         $A \leftarrow A \cup \{G\}$

D.2 Optimality Proof

Definition D.12 (Compatibility). Two executions $G_1$ and $G_2$ are compatible over a program $P$, written $G_1 \#_P G_2$, if:

$$G_1 \#_P G_2 \iff \exists G'_1, G'_2, E_1, E_2, r_1, r_2, s_1, s_2. G_1 \Rightarrow_{nr}^* G'_1 \land \text{next}_P(G'_1) = \bot \land E_1 = G'_1.E$$
$$\land G_2 \Rightarrow_{nr}^* G'_2 \land \text{next}_P(G'_2) = \bot \land E_2 = G'_2.E$$
$$\land G_1.E \cup E_1 = G_2.E \cup E_2$$
$$\land G_1.\text{rf} \cup r_1 = G_2.\text{rf} \cup r_2$$
$$\land G_1.\text{po} \cup s_1 = G_2.\text{po} \cup s_2$$
$$\land s_1 \subseteq (G_1.E \cup E_1) \land s_2 \subseteq (G_2.E \cup E_2) \times E_2$$

Two executions are orthogonal over a program $P$, written $G \perp_P G'$, if they are incompatible: $G \perp G' \iff \neg(G \#_P G')$.

Lemma D.13. Compatibility relation is an equivalence (reflexive, symmetric and transitive) relation.
For all $G_1, G_2, G_3$, if $G_1 \leq_P G_2$ and $G_1 \perp_P G_3$, then $G_2 \perp_P G_3$, where:

$$G \leq_P G' \iff \exists G'', E, r, s. G \Rightarrow_{nr}^* G'' \land \text{next}_P(G'') = \bot \land E = G''.E$$
$$\land G.E \cup E = G''.E \land G.\text{rf} \cup r = G''.\text{rf} \land G.\text{po} \cup s = G''.\text{po}$$
$$\land s \subseteq G''.E \times E$$

In what follows, when the choice of program $P$ is clear from the context, we drop the $P$ subscript and simply write $G_1 \perp G_2$ for $G_1 \perp_P G_2$; and write $G_1 \leq G_2$ for $G_1 \leq_P G_2$.

Definition D.14 (Instrumented configuration). An instrumented configuration is a tuple $\langle G, T, U, S, A \rangle$, where $G, T, U$ and $S$ are defined as before, and $A$ is a set of consistent executions.

Let us instrument our configurations such that an instrumented configuration additionally carries a set $A$ of all the executions generated so far. Similarly, we instrument the implementation in Algorithm 1, Algorithm 2 and Algorithm 3 such that they operate on these instrumented configurations. We thus extend Algorithm 2 as in Algorithm 4 by adding the highlighted line, recording the generated execution in $A$.

Definition D.15 (Instrumented construction). Given a program $P$, an instrumented configuration $\langle G, T, U, S, A \rangle$ is ordered before $\langle G', T', U', S', A' \rangle$ in the instrumented order, written $(G, T, U, S, A) \triangleright_p (G', T', U', S', A')$, if a step of the instrumented algorithm for $P$ transforms $(G, T, U, S, A)$ to $\langle G', T', U', S', A' \rangle$.

When the choice of $P$ is clear from the context, we drop the $P$ subscript and simply write $\triangleright$ for $\triangleright_p$.

Lemma D.16. For all $\langle G, T, U, S, A \rangle$ and $\langle G', T', U', S', A' \rangle$:

$$\text{inv}(\langle G, T, U, S, A \rangle) \land \langle G, T, U, S, A \rangle \triangleright \langle G', T', U', S', A' \rangle \Rightarrow \text{inv}(\langle G', T', U', S', A' \rangle)$$
where

\[ \text{inv}(\langle G, T, U, S, A \rangle) \triangleq \]

\[
\forall r, S[r] \cup U[r] \neq \emptyset \implies r \in G.E
\]

\[
\land \forall (w, r) \in G.rf, r \in T \land (r, w) \in G.pred \implies
\]

\[
\exists P_w, P_w = \text{sbrf}(G', r, w) \land (w, P_w) \in U'[r] \land \text{new}(G, r, w, P_w) \leq G'
\]

\[
\land \forall r_1, r_2, w_1, (w_1, r_1) \in G'.rf \land (r_1, r_2) \in G'.pred \land (r_2 \in T \lor S'[r_2] \neq \emptyset) \implies
\]

\[
(G'.\text{porf}; [w_1]) \times \{r_2\} \subseteq G'.pred
\]

\[
\land \forall G_1, G_2 \in A \cup \{G\}, G_1 \neq G_2 \implies G_1 \perp G_2
\]

\[
\land \forall G_a \in A. \forall r. \forall (w, P_w) \in S'[r]. \text{new}(G, r, w, P_w) \perp G_a
\]

\[
\land \forall G_a \in A. \forall r. \forall (w, P_w) \in S'[r]. \text{new}(G, r, w, P_w) \perp G_a
\]

\[
\land \forall r, (w, P_w) \in S'[r]. G'.rf^{-1}(r) \neq w \lor \text{sbrf}(G', r', w) \perp P_w
\]

with

\[ \text{new}(G, r, w, P_w) \triangleq \text{Combine}(G', P_w).\text{rf}[r \mapsto w] \text{ with } G' = \text{Remove}(G, \text{rng}([r]; G.pred)) \]

\[ \text{sbrf}(G, r, w) \triangleq G_{\text{dom}(G'.\text{porf}; [w]) \setminus \text{dom}(G'.\text{pred}; [r])} \]

\[ \text{cut}(G, r, w) \triangleq \text{Remove}(G, \text{rng}([r]; G'.\text{pred})) \]

Proof. Pick arbitrary \( \langle G, T, U, S, A \rangle, \langle G', T', U', S', A' \rangle \) such that \( \text{inv}(\langle G, T, U, S, A \rangle) \) holds and \( \langle G, T, U, S, A \rangle \rightarrow \langle G', T', U', S', A' \rangle \). There are then three cases to consider: 1) \( \text{next}_P(G) \in R \); or 2) \( \text{next}_P(G) \in W \); or 3) \( \text{next}_P(G) = \perp \).

Case 1.

Let \( r = \text{next}_P(G) \in R \). We then know \( \langle G', T', U', S', A' \rangle = \text{VisitRead}(G, T, U, S, A, r) \) with \( G_r = \text{Add}(G, r) \). Let \( W = W_{\text{cons}} \cup W_{\text{ext}} \) as defined in the algorithm. From the instrumented algorithm we then know there exist \( w_0 \in W \) such that \( G' = G_r, rf^r[r \mapsto w_0], T' = T \cup \{r\}, A' = A, U' = U, \) and \( S'[r'] = S[r'] \) for all \( r' \neq r \) and \( S'[r] = \{(w, \emptyset) | w \in W \setminus \{w_0\}\} \). We are then required to show:

\[ \forall r'. S'[r'] \cup U'[r'] \neq \emptyset \implies r' \in G'.E \]  

(1)

\[ \forall (w, r') \in G'.rf, r' \in T' \land (r', w) \in G'.pred \implies \]

\[ \exists P_w, P_w = \text{sbrf}(G', r', w) \land (w, P_w) \in U'[r'] \land \text{new}(G', r', w, P_w) \leq G' \]  

(2)

\[ \forall r_1, r_2, w_1, (w_1, r_1) \in G'.rf \land (r_1, r_2) \in G'.pred \land (r_2 \in T' \lor S'[r_2] \neq \emptyset) \implies \]

\[ (G'.\text{porf}; [w_1]) \times \{r_2\} \subseteq G'.pred \]  

(3)

\[ \forall G_1, G_2 \in A \cup \{G\}, G_1 \neq G_2 \implies G_1 \perp G_2 \]  

(4)

\[ \forall G_a \in A. \forall r. \forall (w, P_w) \in S'[r]. \text{new}(G, r, w, P_w) \perp G_a \]  

(5)

\[ \forall G_a \in A. \forall r. \forall (w, P_w) \in S'[r]. \text{new}(G, r, w, P_w) \perp G_a \]  

(6)

\[ \forall r', (w, P_w) \in S'[r']. G'.rf^{-1}(r') \neq w \lor \text{sbrf}(G', r', w) \perp P_w \]  

(7)

Part (1) follows immediately from the definitions of \( G', S', U' \) and \( \text{inv}(\langle G, T, U, S, A \rangle) \).

Part (2) follows from the definitions of \( G', U' \), the \( \text{inv}(\langle G, T, U, S, A \rangle) \), that \( (w_0, r) \in G'.pred \), that \( G \leq G' \), and that for all \( w, P_w \) and \( r' \neq r \), \( \text{new}(G', r', w, P_w) = \text{new}(G, r', w, P_w) \).

Part (3) follows from the definitions of \( G', S' \), \( \text{inv}(\langle G, T, U, S, A \rangle) \) and the fact that \( r \) is the maximal element in \( G'.pred \).

For (4), as \( \text{inv}(\langle G, T, U, S, A \rangle) \) holds, it suffices to show \( \forall G_1 \in A. G' \perp G_1 \). Pick an arbitrary \( G_1 \in A \); from the definition of \( G' \) we then have \( G \leq G' \). As such, since from \( \text{inv}(\langle G, T, U, S, A \rangle) \) we have \( G \perp G_1 \), from Lemma D.13 we have \( G' \perp G_1 \) as required.

For (5), pick arbitrary \( G_a \in A, r' \) and \( (w, P_w) \in S'[r'] \). There are then two cases to consider: i) \( r' \neq r \); ii) \( r' = r \). In case (i), we know that \( (w, P_w) \in S'[r'] \) and thus since \( r \) is the maximal (in \( \text{pred} \) order) in \( G' \), we know \( \text{new}(G', r', w, P_w) = \text{new}(G, r', w, P_w) \) and thus from \( \text{inv}(\langle G, T, U, S, A \rangle) \) we have \( \text{new}(G', r', w, P_w) \perp G_a \), as required. In case (ii), we have \( P_w = \emptyset \).

From the definition of \( G' \) we have \( G \leq \text{new}(G', r', w, P_w) \). As such, since from \( \text{inv}(\langle G, T, U, S, A \rangle) \) we have \( G \perp G_a \), from Lemma D.13 we have \( \text{new}(G', r', w, P_w) \perp G_a \), as required.

For (6), pick arbitrary \( G_a \in A, r' \in T' \cap G'.E \) and \( w \) such that \( (w, r') \in G_a.rf \). There are then two cases to consider: i) \( r' \neq r \); ii) \( r' = r \). In case (i), we know \( r' \in T \cap G.E \). As such, from \( \text{inv}(\langle G, T, U, S, A \rangle) \) we have \( (w, r') \in G.a.rf \lor \text{cut}(G', r') \perp G_a \lor \text{new}(G, r', w, P_w) \leq G_a \). That is, since \( U' = U \), from the definition of \( G' \) we
have \((w, r') \in G'.\text{pred} \lor \text{cut}(G, r') \perp G_a \lor \exists P_w. \ (w, P_w) \in U'[r'] \land \text{new}(G, r', w, P_w) \leq G_a\). Moreover, since \(r\) is the maximal (in \text{pred} order) in \(G'\), we know that \(\text{cut}(G, r') = \text{cut}(G', r')\) and \(\text{new}(G', r', w, P_w) = \text{new}(G', r', w, P_w)\). We thus have \((w, r') \in G'.\text{pred} \lor \text{cut}(G', r') \perp G_a \lor \exists P_w. \ (w, P_w) \in U'[r'] \land \text{new}(G', r', w, P_w) \leq G_a\), as required. In case (ii), from the definition of \(G'\) we have \(G = \text{cut}(G', r')\). As such, since from \(\text{inv}(\langle G, T, U, S, A \rangle)\) we have \(G \perp G_a\), we have \(\text{cut}(G', r') \perp G_a\), as required.

For (7), pick arbitrary \(w, P_w, r'\) such that \((w, P_w) \in S'[r']\). There are two cases to consider: i) \(r' \neq r\); ii) \(r' = r\). In case (i), we know \((w, P_w) \in S[r]\). From the definition of \(G'\) we thus have: \(G.f^{-1}(r') = G'.f^{-1}(r')\) and \(\text{sbrf}(G, r', w) = \text{sbrf}(G', r', w)\) and thus the desired result follows immediately from \(\text{inv}(\langle G, T, U, S, A \rangle)\). In case (ii), from the definition of \(G'\) we know that \(w \neq w_0\), as required.

Case 2.

Let \(w = \text{next}_{T}(G) \in W\). We then know \(\langle G', T', U', S', A' \rangle = \text{VisitWrite}(G_w, T, U, S, A, r)\) with \(G_w = \text{Add}(G, w)\). Let set \(R\) be as defined in the algorithm. From the instrumented algorithm we then know \(G' = G_w, T' = T, A' = A\), and for all \(r\), when \(r \notin R\) we have \(S'[r] = S[r]\) and \(U'[r] = U[r]\); and when \(r \in R\) we have \(S'[r] = S[r] \cup \langle w, P_w \rangle\) and \(U'[r] = U[r] \cup \langle w, P_w \rangle\) with \(P_w = \text{sbrf}(G, w, r)\). We are then required to show:

\[
\forall r'. \ S'[r'] \cup U'[r'] \neq \emptyset \implies r' \in G'.E
\]

\[
\forall (w', r') \in G'.\text{rf} \land r' \in T' \land (w, r') \in G'.\text{pred} \implies
\exists P_{w'}. \ (w', P_{w'}) = \text{sbrf}(G', r', w') \land \text{new}(G, r', w', P_{w'}) \leq G'
\]

\[
\forall r_1, r_2, w_1, (w_1, r_1) \in G'.\text{rf} \land (r_1, r_2) \in G'.\text{pred} \land (r_2 \in T' \lor S'[r_2] \neq \emptyset) \implies
(G'.\text{port}^2; [w_1]) \times \{r_2\} \subseteq G'.\text{pred}
\]

\[
\forall G_1, G_2 \in A \cup \{G'\}, \ G_1 \neq G_2 \implies G_1 \perp G_2
\]

\[
\forall G_a \in A. \ \forall r. \ (w', P_w) \in S'[r]. \ \text{new}(G, r, w', P_w) \perp G_a
\]

\[
\forall G_a \in A. \ \forall r \in T' \cap G'.E. \ \forall w'. \ (w', r) \in G_a. \text{rf} \implies
(w', r) \in G'.\text{pred} \lor \text{cut}(G', r) \perp G_a \lor \exists P_{w'}. (w', P_{w'}) \in U'[r] \land \text{new}(G, r, w', P_{w'}) \leq G_a
\]

\[
\forall r, (w', P_{w'}) \in S'[r]. \ G'.f^{-1}(r') \neq w' \lor \text{sbrf}(G', r, w') \perp P_w
\]

Part (8) follows immediately from the definitions of \(G', S', U\) and \(\text{inv}(\langle G, T, U, S, A \rangle)\).

Part (9) follows from the definitions of \(G', U\), the \(\text{inv}(\langle G, T, U, S, A \rangle)\), the fact that \(\neg \exists r. (w, r) \in G'.\text{rf}\), that \(G \leq G'\), and the fact that and that for all \(r, P_w\) and \(w' \neq w\), \(\text{new}(G, r, w', P_w) = \text{new}(G', r, w', P_w)\).

Part (10) follows from the definitions of \(G', S\), \(\text{inv}(\langle G, T, U, S, A \rangle)\), the fact that for all \(r', w', P_{w'}\) if \((w', P_{w'}) \in S'[r'] \setminus S[r']\) then \(r' \in T\), and the fact that for all \(r_1\) we have \((w_1, r_1) \notin G'.\text{rf}\).

For (11), as \(\text{inv}(\langle G, T, U, S, A \rangle)\) holds, it suffices to show \(\forall G_a \in A. \ G' \perp G_a\). Pick an arbitrary \(G_a \in A\); from the definition of \(G'\) we then have \(G \leq G'\). As such, since from \(\text{inv}(\langle G, T, U, S, A \rangle)\) we have \(G \perp G_a\), from Lemma D.13 we have \(G' \perp G_a\) as required.

For (12), pick arbitrary \(G_a \in A, r\) and \((w', P_{w'}) \in S'[r]\). There are two cases to consider: i) \((w', P_{w'}) \in S[r]\); or ii) \((w', P_{w'}) \in S'[r] \setminus S[r]\) and thus \(r \in T \cap G.E. \ w' = w, P_w = \text{sbrf}(G', r, w)\) and \((w, P_w) \notin U[r]\). In case (i), since \(w\) is maximal (in \text{pred} order) in \(G'\), we know \(\text{new}(G, r, w, P_{w'}) = \text{new}(G', r, w', P_{w'})\) and thus from \(\text{inv}(\langle G, T, U, S, A \rangle)\) we have \(\text{new}(G', r, w', P_{w'}) \perp G_a\), as required.

In case (ii), let \(G'' = \text{new}(G', r, w, P_{w'}).\) We know that either \(r \notin G_a.E\) or \(r \in G_a.E\). In the former case, since \(G_a\) is a completed graph and thus \(\text{next}_{T}(G_a.E) = \bot\), by definition we simply have \(G_a \perp G''\), as required. In the latter case, pick \(w''\) such that \((w'', r) \in G_a.\text{rf}\). Since \(r \in T \cap G.E\), from \(\text{inv}(\langle G, T, U, S, A \rangle)\) we have \((w'', r) \in G'.\text{pred} \lor \text{cut}(G', r) \perp G_a \lor \exists P_{w'}. (w', P_{w'}) \in U[r] \land \text{new}(G, r, w', P_{w'}) \leq G_a\). In case of the first disjunct, since \(w \notin G.E\), we have \(w'' \neq w\), i.e., \(G_a.\text{rf}^{-1}(r) \neq G''.\text{rf}^{-1}(r)\), and thus \(G'' \perp G_a\), as required. In case of the second disjunct, from the definition of \(G''\) we have \(\text{cut}(G, r) \leq G''\). As such, from Lemma D.13 we have \(G'' \perp G_a\), as required. In case of the last disjunct, since \((w, P_{w'}) \notin U[r]\) and \((w'', P_{w''}) \in U[r]\), we know that either \(w'' \neq w\) or \(w'' = w \land P_{w''} \neq P_{w}\). In the former case we then have \(G_a.\text{rf}^{-1}(r) \neq G''.\text{rf}^{-1}(r)\) and thus \(G'' \perp G_a\), as required. It is straightforward to demonstrate that as \(w\) is the \text{port} maximal event in both \(P_w\) and \(P_{w''}\) since \(P_{w''} \neq P_w\), we also have \(P_{w''} \neq P_w\). As such, we have \(\text{new}(G, r, w, P_{w''}) \perp \text{new}(G, r, w, P_w)\). Moreover, since \(w'' = w\) and \(\text{new}(G, r, w', P_{w'}) \leq G_a\), we have \(\text{new}(G, r, w, P_{w'}) \leq G_a\). Consequently, from Lemma D.13 we have \(\text{new}(G, r, w, P_{w'}) \perp G_a\). On the other hand, from the definition of \(G'\) we have \(\text{new}(G, r, w, P_{w}) = \text{new}(G', r, w, P_{w})\). As such, we have \(\text{new}(G, r, w, P_{w}) \perp G_a\), as required.
For (13), pick arbitrary \( G_a \in A, r \in T' \cap G'.E \) and \( w' \) with \( (w', r) \in G_a.rf \). From \( \text{inv}(G, T, U, S, A) \) we have \((w', r) \in G.a.pred \lor \text{cut}(G, r) \perp G_a \lor \bigvee w. (w', P_w) \in U[r] \land \text{new}(G, r, w, P_w) \leq G_a. \) From the definition of \( G' \) and since \( w \) is maximal (in \( G'.pred \) order), we know that \( \text{cut}(G, r) = \text{cut}(G', r) \) and \( \text{new}(G, r, w, P_w) = \text{new}(G', r, w, P_w) \). As such, since \( U[r] \subseteq U'[r] \), from the definition of \( G' \) we have \((w', r) \in G'.pred \lor \text{cut}(G', r) \perp G_a \lor \bigvee w. (w', P_w) \in U'[r] \land \text{new}(G, r, w, P_w) \leq G_a, \) as required.

For (14), pick arbitrary \( w', P_w, r \) such that \((w', P_w) \in S'[r] \). There are two cases to consider: i) \((w', P_w) \in S[r]; \) ii) \((w', P_w) \in S'[r] \setminus S[r] \) and thus \( w' = w. \) In case (i), since \( w \) is maximal (in \( G'.pred \) order), we have \( \text{sbrf}(G, r, w) \leq \text{sbrf}(G', r, w'). \) As such, the desired result follows from \( \text{inv}(G, T, U, S, A) \) and Lemma D.13. In case (ii), since \( w \notin G \), we know that \( G.rf^{-1}(r) \neq w. \) As such, from the definition of \( G' \) we have \( G'.rf^{-1}(r) = G.rf^{-1}(r) \neq w, \) as required.

Case 3.

Let \( U = \text{next}_p(G) \). We then know \( (G', T', U', S', A') = \text{RevisitReads}(G, T, U, S, A) \). That is, there exists \( w, P_w \) such that \( r \in G.E \cdot (w, P_w) \in S[r]; \) for all \( r' \) if \( (r, r') \in G.a.pred \) then \( S[r'] = \emptyset; G' = \text{new}(G, r, w, P_w); T' = T \setminus P_w.E; A' = A \cup \{ G \}; S'[r] = S[r] \setminus \{(w, P_w)\}; \) for all \( r' \neq w \) we have \( S'[r'] = S'[r]; \) for all \( r' \) such that \( (r, r') \in G.a.pred \) we have \( U'[r'] = \emptyset; \) and for all \( r' \) such that \( (r', r) \in G.a.pred \) we have \( U'[r'] = U[r'] \). We are then required to show:

\[
\forall r' \cdot S'[r'] \cup U'[r'] \neq \emptyset \Rightarrow r' \in G'.E \tag{15}
\]

\[
\forall (w', r') \in G.a.rf \land (r', w') \in G'.a.pred \Rightarrow \exists P_{w'}. P_{w'} = \text{sbrf}(G', r', w') \land (w', P_w) \in U'[r'] \land \text{new}(G', r', w', P_w) \leq G'. \tag{16}
\]

\[
\forall r_1, r_2, w_1, (w_1, r_1) \in G.a.rf \land (r_1, r_2) \in G'.a.pred \land (r_2 \in T' \lor S'[r_2] \neq \emptyset) \Rightarrow (G'.a.porf^2; [w_1]) \times \{ r_2 \} \subseteq G'. \tag{17}
\]

\[
\forall G_1, G_2 \in A \cup \{ G \} \cup \{ G' \}. G_1 \neq G_2 \Rightarrow G_1 \perp G_2 \tag{18}
\]

\[
\forall G_a \in A \cup \{ G \}. \forall r' \cdot (w', r') \in G.a.rf \Rightarrow (w', r') \in G'.a.rf \tag{19}
\]

\[
\forall r', (w', P_w') \in S'[r']. G'.rf^{-1}(r') \neq w \lor \text{sbrf}(G', r', w') \perp P_w' \tag{20}
\]

Part (15) follows immediately from the definitions of \( G', S', U', \text{inv}(G, T, U, S, A) \) and the maximality of \( r \) in \( S \).

For part (16), pick an arbitrary \((w', r') \in G'.a.rf \) such that \( r' \in T' \) and \( (r', w') \in G'.a.pred \). Since \( r' \in T' \) we know \( r' \notin P_w \) and thus \((r', r) \in G'.pred \), and thus \((r', r) \in G'.rf \), and thus the definition of \( G' \) we know \((r', r) \in G.a.rf \). There are now two cases to consider: i) \((r', r) \in G.a.rf \); ii) \((r', r) \in G'.a.rf \). Since \( (w', r') \in G.a.rf \) we know \( (w', r') \in G'.a.rf \). Moreover, since \( (w', w') \in U[r], \) from the definition of \( U' \) we know \( (w, P_w) \in U'[r] \). We then know there exists \( w' = w \) and \( P_w = P_w' \) such that \( \text{sbrf}(G', r', w') = P_w', (w', P_w') \in U'[r] \) and \( G' = \text{new}(G', r, w, P_w') \leq G'. \) As required.

In case (2), since \((r', r) \in G'.pred \) and \( r' \notin P_w \), we have \((r', r) \in G'.pred \). As \((w', r') \in G'.rf \), from the definition of \( G' \) we know \((w', r') \in G'.rf \). Moreover, since \( S'[r] \neq \emptyset \) and \( (w, P_w) \in S[r], \) from \( \text{inv}(G, T, U, S, A) \) we know \( (G'.a.porf^2; [w]) \times \{ r \} \subseteq G'.pred \). That is, \((w, r) \in G'.pred \) and \((w, r') \in G'.pred \), and \( \text{sbrf}(G', r', w') \subseteq \text{sbrf}(G', r', w') \). Since \((w, r) \in G'.a.rf \) and \((w, r') \in G'.a.rf \), on the other hand, since \( r' \in T' \), from the definition of \( T \) we know \( r \in T. \) As such, since \((w', r') \in G'.a.rf \), we have \((r', r) \in G'.pred \). From \( \text{inv}(G, T, U, S, A) \) we know there exists \( P_{w'} \) such that \( P_{w'} = \text{sbrf}(G', r', w'), \) \((w', P_w') \in U'[r] \land \text{new}(G', r', w', P_w') \leq G'. \)

For part (17), pick arbitrary \( w_1, r_1, r_2 \) such that \((w_1, r_1) \in G'.a.rf \), \((r_1, r_2) \in G'.pred \) and \( r_2 \in T' \lor S'[r_2] \neq \emptyset. \) We first demonstrate that \((r_2, r) \in G'.pred \). Given the disjunction \( r_2 \in T' \lor S'[r_2] \neq \emptyset, \) if the first disjunct holds, since \( r_2 \in G'.E \) and from the definition of \( T' \) we know \( T' \cap P_w.E = \emptyset, \) from the definition of \( G' \) we have \((r_2, r) \in G'.pred \). If however the second disjunct holds, from the definition of \( S' \) and since \( r \) is picked to be maximal in \( S \), we have \((r_2, r) \in G'.pred \). Since \((r_1, r_2) \in G'.pred \) and thus \((r_2, r) \in G'.pred \), from the definition of \( G' \) we know \((r_1, r_2) \in G'.pred \) and thus \((r_2, r) \in G'.pred \). Moreover, since \((w_1, r_1) \in G'.a.rf \), from the definition of \( G' \) we know \((w_1, r_1) \in G'.a.rf \). Consequently, since \( r_2 \in T' \lor S'[r_2] \neq \emptyset, T' \subseteq T \) and \( S'[r_2] \subseteq S[r_2], \) from \( \text{inv}(G, T, U, S, A) \) we have \((G'.a.porf^2; [w_1]) \times \{ r_2 \} \subseteq G'.pred \). As such, since \((r_2, r) \in G'.pred \), from the definition of \( G' \) we have \((G'.a.porf^2; [w_1]) \times \{ r_2 \} \subseteq G'.pred. \) Moreover, from
the definition of $G'$ and since $(G, \text{porf}; \{w_1\}) \times \{r_2\} \subseteq G'.\text{pred}$, and $(r_2, r) \in G.\text{pred}$ (thus $(r_2, r) \in G'.\text{pred}$), we know $(G, \text{porf}; \{w_1\}) = (G'.\text{porf}; \{w_1\})$. That is, we have $(G'.\text{porf}; \{w_1\}) \times \{r_2\} \subseteq G'.\text{pred}$, as required.

For (18), as $\text{inv}(G, T, U, S, A)$ holds, it suffices to show: i) $G' \perp G$ and ii) $G' \perp G_a$ for all $G_a \in A$. Part (i) follows from the definition of $G'$ and the last conjunct of $\text{inv}(G, T, U, S, A)$. Part (ii) follows from the definition of $G'$ and the fifth conjunct of $\text{inv}(G, T, U, S, A)$.

For (19), pick an arbitrary $G_a \in A \cup \{G\}$, $r'$ and $(w', P'_w) \in S'[r']$. From the definition of $S'$ we then have $(w', P'_w) \in S[r']$. Moreover, as $r$ is the maximal entry in $S$, we know that $(r', r) \in G'.\text{pred}$, and thus from the definition of $G'$ we have new$(G', r', w', P'_w) = \text{new}(G, r', w', P'_w)$. There are now two cases to consider: 1) $G_a \in A$; or 2) $G_a = G$. In case (1), from the definition of $G'$ we have new$(G', r', w', P'_w) \perp G_a$, and since new$(G', r', w', P'_w) = \text{new}(G, r', w', P'_w)$, we have new$(G, r', w', P'_w) \perp G_a$, as required.

In case (2), from $\text{inv}(G, T, U, S, A)$ we have $G.\text{rf}^{-1}(r') \neq w' \vee \text{sbrf}(G, r', w') \perp P'_w$, and thus new$(G, r', w', P'_w) \perp G$. As such, since new$(G', r', w', P'_w) = \text{new}(G, r', w', P'_w)$, we have new$(G', r', w', P'_w) \perp G$, as required.

For (20), pick an arbitrary $G_a \in A \cup \{G\}$, $r' \in T' \cap G.E$ and $w'$ such that $(w', r') \in G_a.\text{rf}$. From the definition of $T'$ and $G'$ we then have $r' \in T \cap G.E$ and $w' \notin P_w$, and since $r$ is the maximal entry in $S$, we know that $(r', r) \in G'.\text{pred}$. There are now two cases to consider: 1) $G_a \in A$; or 2) $G_a = G$.

In case (1), from $\text{inv}(G, T, U, S, A)$ we have $(w', r') \in G.\text{pred} \vee \text{cut}(G, r') \perp G_a \lor \exists P'_w (w', P'_w) \in U[r'] \land \text{new}(G, r', w', P'_w) \subseteq G_a$. From the definition of $G'$ we have cut$(G', r') = \text{cut}(G, r')$ and new$(G', r', w', P'_w) = \text{new}(G, r', w', P'_w) \perp G_a$. Moreover, from the definition of $U'$ and since $(r', r) \in G'.\text{pred}$, we have $U'[r'] = U[r']$. As such, from the definition of $G'$ we have $(w', r') \in G'.\text{pred} \land \text{cut}(G', r') \perp G_a \lor \exists P'_w (w', P'_w) \in U'[r'] \land \text{new}(G', r', w', P'_w) \subseteq G_a$, as required.

In case (2), there are two cases to consider: either $(w', r') \in G.\text{pred}$; or $(w', r') \notin G.\text{pred}$. In the former case from the definition of $G'$ we have $(w', r') \in G'.\text{pred}$, as required. In the latter case, from $\text{inv}(G, T, U, S, A)$ we know there exists $P'_w$ such that $P'_w = \text{sbrf}(G, r', w')$, $(w', P'_w) \in U[r']$ and new$(G, r', w', P'_w) \perp G$. From the definition of $U'$ and since $(r', r) \in G'.\text{pred}$, we have $U'[r'] = U[r']$. Moreover, from the definition of $G'$ we have new$(G', r', w', P'_w) = \text{new}(G, r', w', P'_w) \perp G$. As such, we know there exists $P'_w$ such that $(w', P'_w) \in U'[r']$ and new$(G', r', w', P'_w) \perp G$, as required.

For (21), pick arbitrary $w', P'_w$ such that $(w', P'_w) \in S'[r']$. There are now two cases to consider: 1) $r' = r$; or 2) $r' \neq r$ and thus since $r$ is chosen to be pred-maximal in $G$, we have $(r', r) \in G.\text{pred}$. In case (1), since $(w, P_w), (w', P'_w) \in S[r]$ and $S[r]$ is a set, we know that $w \neq w' \lor P_w \neq P'_w$. On the other hand, from the definition of $G'$ we have $\text{sbrf}(G', r, w) = P_w$. It is then straightforward to demonstrate that $P'_w \neq \text{sbrf}(G', r, w) \Rightarrow P'_w \perp \text{sbrf}(G', r, w)$. As such, we have $w \neq w' \lor \text{sbrf}(G', r, w) \perp P'_w$, as required.

In case (2), from $\text{inv}(G, T, U, S, A)$ we have $G.\text{rf}^{-1}(r') \neq w' \lor \text{sbrf}(G', r', w') \perp P'_w$. If the first disjunct holds $(G.\text{rf}^{-1}(r') = w' \lor \text{sbrf}(G', r', w') \perp P'_w)$, since $(r', r) \in G.\text{pred}$, from the definition of $G'$ we have $G.\text{rf}^{-1}(r') = G'.\text{rf}^{-1}(r') \neq w'$, as required.

On the other hand, if the first disjunct does not hold and instead the second disjunct holds, we then have $G.\text{rf}^{-1}(r') = w' \lor \text{sbrf}(G', r', w') \perp P'_w$. Since $(w', r') \in G.\text{rf}$, $(r', r) \in G.\text{pred}$ and $S[r] \neq \emptyset ((w, P_w) \in S[r])$, from $\text{inv}(G, T, U, S, A)$ we have $G.\text{porf}; \{w'\} \times \{r\} \subseteq G.\text{pred}$. As such, from the definition of $G'$ we know $w' \in G'$ and $\text{sbrf}(G', r', w') = \text{sbrf}(G', r', w')$. Consequently, since $\text{sbrf}(G', r', w') \perp P'_w$, we have $\text{sbrf}(G', r', w') \perp P'_w$, as required.

Theorem D.17 (Optimality). For all $G, T, U, A$, if $\langle G_0, \emptyset, \emptyset, \emptyset, \emptyset \rangle \Rightarrow^* \langle G, T, U, \emptyset, A \rangle$ then:

$$\forall G_1, G_2 \in A \cup \{G\}, G_1 \neq G_2 \Rightarrow G_1 \perp G_2$$

Proof. From the definition of $\text{inv}(\cdot)$, the $\text{inv}(G_0, \emptyset, \emptyset, \emptyset, \emptyset)$ holds vacuously. The desired result then follows from Lemma D.16 by straightforward induction on the number of steps in $\Rightarrow^*$. □