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 equivalence class on executions.

1 Introduction

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

$$\begin{array}{l|l} x := 1 & a := y \\ y := 1 & b := x \\ & \textbf{assert}(a \leq b) \end{array} \qquad (\text{MP})$$

and we want to check whether its assertions are always satisfied. To do this, we can simply enumerate all executions of the program, and check each execution individually. An effective way of doing so is using *stateless model checking* [17, 18, 32]. There are, however, 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 [25] and TSO [34], but not under PSO [38], or C11 with 'relaxed' accesses [8].

The second challenge is due the *exponential* number of executions that need to be explored for any non-trivial concurrent program. To tackle this, *partial order reduction* techniques [1, 11, 15, 19, 40] 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, 11, 14, 15, 19–21, 23, 33, 37, 40], a *generic* technique that combats both these challenges is yet to be developed.[1]

[1]The only generic tool, Herd [6], is not competitive in terms of performance.

Conference'17, July 2017, Washington, DC, USA
2019. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

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 locks, barriers, and FIFO queues, 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 three basic assumptions: sbrf-acyclicity, extensibility, and prefix-closedness (§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 (§5.3).
- 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.
- We implement GenMC into a tool for verifying C programs, and demonstrate that it has comparable or better performance than the state-of-the-art specialized tools for specific memory models (§5).

2 Overview

In the literature of axiomatic memory models [6, 24], the traces of shared memory accesses generated by a program are commonly 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 *sequenced-before* (sb, a.k.a. the program order) 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 certain properties prescribed by a *consistency* predicate.

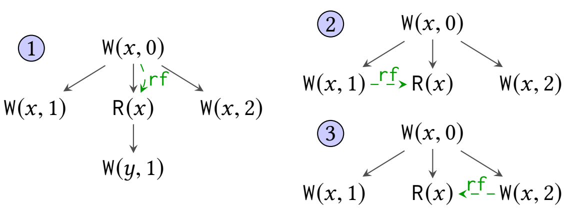


Figure 1. Execution graphs of W+RW+W.

For example, consider the following program:[2]

$$x := 1 \;\Big\|\; \begin{array}{l} a := x; \\ \textbf{if } a = 0 \textbf{ then } y := 1 \end{array} \;\Big\|\; x := 2 \qquad (\text{W+RW+W})$$

Under SC [25], 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] and cppmem [8], is to enumerate *all* possible executions and filter them according to the consistency predicate of the memory model.

A problem with this approach is that under an arbitrary memory model, the set of graphs generated may be infinite! To see this, consider the following program:

$$x := y \;\|\; y := x \qquad (\text{LB+DEP})$$

Without any constraints on the memory model, the program can return $x = y = v$, for *any* value v, by having both threads read v and write v as follows:

$$\begin{tikzcd} \mathtt{R}(y) \arrow[d] & & \mathtt{R}(x) \arrow[d] \\ \mathtt{W}(x, v) \arrow[urr, dashed, "\mathsf{rf}"] & & \mathtt{W}(y, v) \arrow[ull, dashed] \end{tikzcd}$$

In the weak memory concurrency literature, such executions are considered problematic as they generate values "out of thin air" [10, 28, 39]. To avoid such cases, we require that the underlying memory model satisfy the following (see §3):

MM1: sbrf is irreflexive, where $\mathsf{sbrf} \triangleq (\mathsf{sb} \cup \mathsf{rf})^+$

This requirement ensures that loop-free programs have only finitely many possible executions.

Remark 1. An alternative way to avoid such problematic executions is for a model to record (syntactic) dependencies in executions and forbid dependency cycles. For simplicity, we avoid recording dependencies and opt for the stricter requirement of sbrf-irreflexivity. Handling models recording dependencies is left for future work.

2.1 Checking Consistency at Every Step

Although the sbrf-irreflexivity requirement precludes the problematic "out of thin air" scenarios, generating *all* executions and then checking consistency does not scale [22].

A much better approach, followed by most tools (e.g., [1, 2, 4, 22, 33]), is to construct executions *incrementally* by adding events one at a time and checking for consistency at each step, thereby avoiding the exploration of inconsistent graphs. For this approach to work, the underlying memory model must satisfy the following condition:

MM2′: Every non-empty consistent graph has an sb-maximal event that, if removed, yields a consistent graph.

This condition ensures that each execution can be generated by adding its events in *some* total extension of the sbrf order, 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: $\mathtt{W}(x, 0)$, $\mathtt{W}(x, 1)$, $\mathtt{R}(x, 1)$, and $\mathtt{W}(x, 2)$.

2.2 Fixing the Graph Construction Order

To generate all executions of a program, following **MM2′**, one must in principle consider *all* possible extensions of sbrf. This, however, very often leads to duplicate explorations.

Therefore, ideally, one would generate all executions without considering all possible extensions of sbrf, regardless of the memory model. In fact, we can do this for all well-known memory models. In particular, models such as SC [25], TSO [34], PSO [38], and RC11 [24] all satisfy an even stronger guarantee than that of **MM2′**, namely *prefix-closedness*:

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 *any* total extension of sbrf.

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, GenMC, to generate the executions of W+RW+W by adding its events in a *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 $\mathtt{W}(x, 0)$ (see below). First, we add the $\mathtt{W}(x, 1)$ write of the first thread to G_0, simultaneously adding the appropriate sb edge between the events:

$$\begin{tikzcd} \mathtt{W}(x, 0) & & & \mathtt{W}(x, 0) \arrow[dl] \\ & \rightsquigarrow & \mathtt{W}(x, 1) & \end{tikzcd}$$

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

$$\begin{tikzcd} & \mathtt{W}(x, 0) \arrow[dl] & & & \mathtt{W}(x, 0) \arrow[dl] \arrow[dr] \arrow[dr, dashed, bend left, "\mathsf{rf}"] & & & \mathtt{W}(x, 0) \arrow[dl] \arrow[dr] & \\ \mathtt{W}(x, 1) & & \rightsquigarrow \Bigg\{ & \mathtt{W}(x, 1) & & \mathtt{R}(x) & \mathtt{W}(x, 1) \arrow[rr, dashed, "\mathsf{rf}"] & & \mathtt{R}(x) \Bigg\} \end{tikzcd}$$

[2]In all our examples, we us x, y, z as global (shared) variables,and a, b, c as local variables. All variables are implicitly initialized to 0.

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., $\mathtt{W}(x, 0)$, and record the alternative(s), i.e., $\mathtt{W}(x, 1)$, 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 $\mathtt{W}(x, 1)$ as *forward* revisits since they are already in the graph when the read ($\mathtt{R}(x)$) is added to the graph.

$$\begin{tikzcd}[column sep=small] & \mathtt{W}(x,0) \arrow[dl] & & & \mathtt{W}(x,0) \arrow[dl] \arrow[dr] \arrow[dr, dashed, "\mathtt{rf}"] & \\ \mathtt{W}(x,1) & & \rightsquigarrow & \mathtt{W}(x,1) & & \mathtt{R}(x) \end{tikzcd} \qquad \begin{array}{c} W[a{:=}x] \\ \boxed{\mathtt{W}(x,1)} \end{array}$$

Since the value read is 0, we next add the $\mathtt{W}(y, 1)$ write of thread 2. Finally, we add $\mathtt{W}(x, 2)$ 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 $\mathtt{W}(x, 2)$ as *backward* revisits since they are added to the graph after the corresponding read ($\mathtt{R}(x)$).

$$\begin{tikzcd}[column sep=small] & \mathtt{W}(x,0) \arrow[dl] \arrow[dr] \arrow[dr, dashed, "\mathtt{rf}"] & & & & \mathtt{W}(x,0) \arrow[dl] \arrow[d] \arrow[d, dashed, "\mathtt{rf}"] \arrow[dr] & \\ \mathtt{W}(x,1) & & \mathtt{R}(x) & \rightsquigarrow & \mathtt{W}(x,1) & \mathtt{R}(x) \arrow[d] & \mathtt{W}(x,2) \\ & & & & & \mathtt{W}(y,1) & \end{tikzcd} \qquad \begin{array}{c} W[a{:=}x] \\ \boxed{\mathtt{W}(x,1)} \\ \boxed{\mathtt{W}(x,2)} \end{array}$$

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

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

$$\begin{tikzcd}[column sep=small] & \mathtt{W}(x,0) \arrow[dl] \arrow[d] \arrow[dr] & \\ \mathtt{W}(x,1) & \mathtt{R}(x) & \mathtt{W}(x,2) \arrow[l, dashed, "\mathtt{rf}"'] \end{tikzcd} \qquad \begin{array}{c} W[a{:=}x] \\ \boxed{\mathtt{W}(x,1)} \\ \boxed{\mathtt{W}(x,2)}\ ✓ \end{array} \tag{Ex2}$$

Note that when considering the alternative reads-from option $\mathtt{W}(x, 2)$, it is essential to restrict the graph to contain only the events added *prior* to the read, as the new value read may affect the control flow. For instance, it is crucial to remove $\mathtt{W}(y, 1)$ as it is only present when 0 is read from x. Similarly, we must retain the events added before (in sbrf order) the alternative reads-from option $\mathtt{W}(x, 2)$. For instance, if $x := 2$ in (W+RW+W) is wrapped in the conditional **if** $y = 1$ **then**, the presence of the $\mathtt{W}(x, 2)$ event in the graph depends on the value read for y, i.e., the events before $\mathtt{W}(x, 2)$ in sbrf order.

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

$$\begin{tikzcd}[column sep=small] & \mathtt{W}(x,0) \arrow[dl] \arrow[dr] & \\ \mathtt{W}(x,1) \arrow[rr, dashed, "\mathtt{rf}"] & & \mathtt{R}(x) \end{tikzcd} \qquad \begin{array}{c} W[a{:=}x] \\ \boxed{\mathtt{W}(x,2)}\ ✓ \end{array} \tag{Pre3}$$

To continue, we add the $\mathtt{W}(x, 2)$ write arriving at the graph below, corresponding to execution ③ in Fig. 1. Note that we do not re-add this write as an entry in W, as this option has already been explored (✓-marked).

$$\begin{tikzcd}[column sep=small] & \mathtt{W}(x,0) \arrow[dl] \arrow[d] \arrow[dr] & \\ \mathtt{W}(x,1) \arrow[r, dashed, "\mathtt{rf}"] & \mathtt{R}(x) & \mathtt{W}(x,2) \end{tikzcd} \qquad \begin{array}{c} W[a{:=}x] \\ \boxed{\mathtt{W}(x,2)}\ ✓ \end{array} \tag{Ex3}$$

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

Avoiding Duplication When revisiting a read, 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 ($\mathtt{W}(x, 1)$), we removed $\mathtt{W}(x, 2)$ arriving at (Pre3). We later re-added $\mathtt{W}(x, 2)$ and obtained (Ex3). In doing so, we did not re-add $\mathtt{W}(x, 2)$ as a (backward) revisit option to W as this option had already been explored before. Rather, by having previously marked $\mathtt{W}(x, 2)$ as explored (✓-marked), we ascertained that $\mathtt{W}(x, 2)$ 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., $\mathtt{R}(x)$), only events added after the read are removed from the graph. As such, since a forward option (e.g., $\mathtt{W}(x, 1)$) 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., $\mathtt{W}(x, 1)$ 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 sbrf order. This is because in cases where a read is added before the write it reads from, e.g., reading from $\mathtt{W}(x, 2)$ in ②, the 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 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

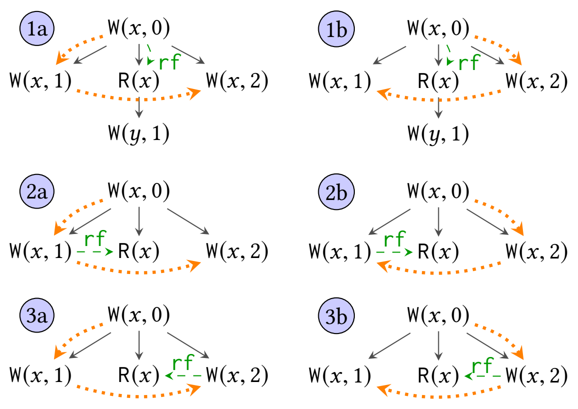


Figure 2. mo-executions of W+RW+W under SC.

0, then there cannot be a read of x that also reads 0":

$$a := x \;\big\|\; b := y \;\big\|\; x := 42 \qquad (\text{R+R+W})$$

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, an sb-maximal event can always be added to G to yield a consistent execution (with an appropriate 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 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 rf.

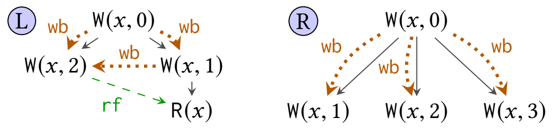
2.5 GenMC: Modification Order and Writes-Before

Recall that using GenMC, we generated all three executions of 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, which we call mo-*executions*. There, mo-executions contain an additional component: the *modification order*, mo, also known as the coherence order, which totally orders all writes to a given memory location.

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

One can of course adapt GenMC to enumerate all mo-executions, as e.g., in [22]; but doing so is wasteful because while the choice of 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.[3]

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



In execution Ⓛ, the W(x, 1) must *write-before* W(x, 2): otherwise, the read may only read 1, due to coherence. Of course, the initialization write writes before the writes of both threads, as it is sb-before them. By contrast, in execution Ⓡ, the writes of the three threads are not wb-ordered, as there is no causal ordering amongst them.

Computing wb can be done in cubic time, and yields a complete procedure for checking consistency for RC11 without SC features. For SC, while checking consistency of an execution is NP-complete [16], a wb-based check can approximate it extremely well[4].

2.6 GenMC: Handling rf-Functionality Constraints

Memory models may prescribe rf-*functionality* constraints requiring that certain writes be read by at most one read. For instance, in case of the RMW (read-modify-write) instructions, e.g., CAS (compare-and-swap) or FAI (fetch-and-increment), to ensure their atomicity, two RMW events may not read from the same write. These constraints are, however, not exclusive to RMWs. For instance, as we discuss in the upcoming section, a lock library may require rf-functionality to ensure mutual exclusion. Indeed, as shown in [36], many well-known concurrent libraries require rf-functionality to ensure correct synchronization.

Handling such constraints requires additional care. Consider the program below and its executions depicted in Fig. 3:

$$a : \mathsf{FAI}(x) \;\big\|\; b : \mathsf{FAI}(x) \qquad (\text{FAI/2})$$

Execution ① captures the case where thread 1 increments x first, while ② captures the case where thread 2 increments x first. Let us run GenMC on this example. We proceed by adding the RMW instruction of thread 1 (a) which reads from the initialization write. When we next add the RMW instruction of thread 2 (b), to ensure atomicity, there is only one consistent option for b to read from, namely a. However, this poses a problem: when b is added, it cannot add b to W as a revisit option for a (since that would create an sbrf cycle). As such, the algorithm fails to generate execution ②.

[3]To see that, consider an extension of W+RW+W with n parallel writes and one reader: that program has $n + 1$ executions and $(n + 1)!$ mo-executions.
[4]For a definition of wb see the technical appendix.

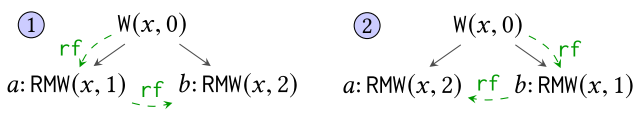
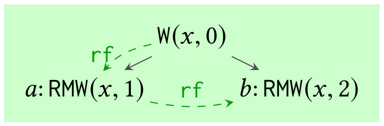


Figure 3. The executions of the FAI/2 program.

To remedy this, we allow for *temporary inconsistency* in the graph. More specifically, we push to W options that break such consistency constraints.

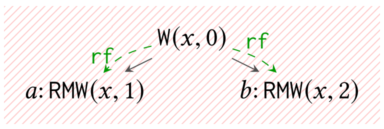
When this inconsistent execution is eventually picked from W, during the course of its exploration, we may encounter events that can revisit one of the events responsible for inconsistency, thus obtaining a consistent graph. The inconsistent execution is then dropped.

In our example, we push to W an entry for b to read from the initial write, and continue with the consistent option:



$W[b]$ $\boxed{W(x,0)}$

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.



$W[a]$ $\boxed{b}$ (FAI-⊥)

Finally, we pick the remaining revisit option in $W[a]$, restrict the graph as before and arrive at execution ②.

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

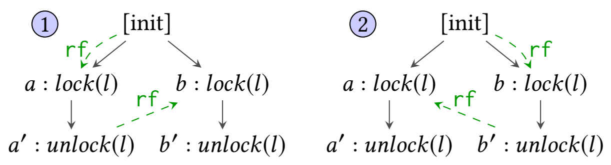


Figure 4. Executions of the LOCK/2 program.

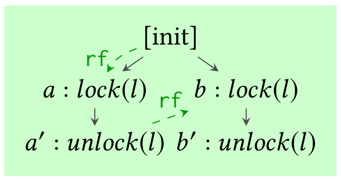
adding a read (lock) event e to the graph, there may not exist a write from which e could read. As such, `rf` is not necessarily total on reads in such libraries. However, lock events may not block arbitrarily: a lock may block only when all writes are already read from; i.e., when the mutex is taken.

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

$$\begin{array}{l||l} a : lock(l); & b : lock(l); \\ a' : unlock(l); & b' : unlock(l); \end{array} \qquad \text{(LOCK/2)}$$

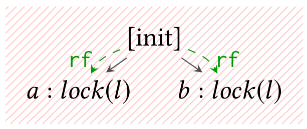
Note that neither lock call may block as the program contains sufficient writes: two unlocks and the implicit initialization.

Running our algorithm on this example, we add the events in order (a, a', b, b') and obtain execution ①. As with FAI/2, when adding b to the graph, we also consider inconsistent reads-from options and add them to the work list, arriving at the following configuration:



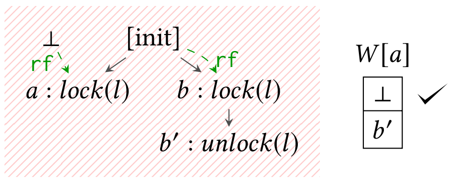
$W[b]$ $\boxed{\text{[init]}}$

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 revisit 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):



$W[a]$ $\boxed{\perp}$ (LOCK-⊥)

We next pick ⊥ as a revisit option for a. Since a is now blocking, its thread cannot proceed and its subsequent events are skipped. We thus next add b' 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 (a is 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**-**MM3**. 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 we present the SC [29], TSO [34] and RC11 [24] 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 $\langle i, n, l\rangle$, where $i \in \mathsf{Tid} \uplus \{0\}$ is a *thread identifier* (0 for initialization events) with $\mathsf{Tid} \subseteq \mathbb{N}$, $n \in \mathbb{N}$ is the *serial number* inside a thread, and $l \in \mathsf{Lab}$ is an event *label*. The serial number of an event denotes its index (from 1) within its thread; e.g., the first event of a thread has serial number 1. Serial number 0 is reserved for initialization events. A label may be either: (i) the *error* label `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 $\mathtt{W}(x, 1)$ 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* ($\mathtt{R}$) and *write* ($\mathtt{W}$) events, an *execution* is a tuple $G{=}\langle E, rf\rangle$, where E is a sequence of *events*, and $rf : E \cap \mathtt{R} \rightharpoonup E \cap \mathtt{W}$ 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 $G.\mathsf{E}$ and $G.\mathtt{rf}$ for its components, and write $G.\mathtt{R}$ (resp. $G.\mathtt{W}$) for $G.\mathsf{E} \cap \mathtt{R}$ (resp. $G.\mathsf{E} \cap \mathtt{W}$). Although $G.\mathtt{rf}$ is a function, we often implicitly coerce it to a relation on $\mathtt{W} \times \mathtt{R}$. We write $G.\mathsf{E}_0$ for $G.\mathsf{E} \cap \mathsf{Event}_0$; we write $G.\mathsf{E}_i$ for $\{\langle i', -, -\rangle \in G.\mathsf{E} \mid i = i'\}$; and write $G.\mathsf{sb}$ for the *sequenced-before* relation defined as follows:

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

In general the rf function may not be *total*; as such, we write $G.\mathsf{B}$ for the set of *blocked events* $\{r \in G.\mathtt{R} \mid r \notin dom(G.\mathtt{rf})\}$. That is, such read events are blocked in that they cannot read a value and thus have no incoming $G.\mathtt{rf}$ edges. As we show shortly, we use this to model blocking library events, e.g., a blocking lock event that is awaiting the release of a mutex.

Notation Given a relation r and a set A, we write $\mathsf{r}^?$, r^+ and r^* for the reflexive, transitive and reflexive-transitive closure of r, respectively. We write $dom(\mathsf{r})$ and $rng(\mathsf{r})$ for the domain and range of r, respectively. We write r^{-1} for the inverse of r; $\mathsf{r}|_A$ for $\mathsf{r} \cap (A \times A)$; and $[A]$ for the identity relation on A: $\{(a, a) \mid a \in A\}$. Given relations r_1 and r_2, we write $\mathsf{r}_1; \mathsf{r}_2$ for $\{(a, b) \mid \exists c.\ (a, c) \in \mathsf{r}_1 \wedge (c, b) \in \mathsf{r}_2\}$, i.e., their relational composition. Given an event set E, we write E_x for $\{e \in E \mid \mathtt{loc}(e){=}x\}$, and $G|_E$ for $\langle E', G.\mathtt{rf}|_{E'}\rangle$ with $E' \triangleq G.\mathsf{E} \cap E$. We write $G.\mathtt{sbrf}$ for $(G.\mathsf{sb} \cup G.\mathtt{rf})^+$, and write $G.\mathtt{rf}[r \mapsto w]$ for the graph obtained from mapping $G.\mathtt{rf}(r)$ to w. Finally, we write $+\!\!+$ for sequence concatenation.

Extension We define graph *extension* in Theorem 3.2, used by the incremental construction in our algorithm, which denotes adding an *available* event to an execution. Given an execution G, an event $\langle i, n{+}1, -\rangle$ is available when $|G.\mathsf{E}_i| = n$; i.e., thread i contains n events. As executions are constructed incrementally, adding one event at a time, it is unnecessary to require that G include adjacent events indexed $1 \cdots n$ for i, as this is ensured by checking availability at each step.

Definition 3.2 (Extension)**.** Given execution G and event $e{=}\langle i, n{+}1, l\rangle$, if $n = |G.\mathsf{E}_i|$, then e is *available* for G. The *extension* of G with an available event e, written $\mathsf{Add}(G, e)$, denotes the execution $\langle E{+\!\!+}[e], G.\mathtt{rf}\rangle$.

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 $\mathsf{cons}_{\mathsf{m}}(G)$ 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**, **MM2** and **MM3**. In what follows, we formally define these conditions.

The first condition (**MM1**) is captured by Theorem 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 $\langle E, rf \rangle$ is consistent then $\langle E', rf \rangle$ is also consistent, where $E' \in \mathsf{perm}(E)$ is a permutation of E.

Moreover, well-formedness requires that if $\mathsf{cons_m}(G)$ holds, then two further conditions (2-3) hold. The first requires that blocking reads be maximal in $G.\mathtt{sbrf}$: if an event blocks then it cannot proceed. The second stipulates that reads block only when all writes are *matched*. That is, if there is a blocking read on x ($G.\mathsf{R}_x \not\subseteq dom(G.\mathtt{rf})$), then all writes on x have already been read-from ($G.\mathsf{W}_x \subseteq rng(G.\mathtt{rf})$). Note that when $G.\mathtt{rf}$ 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 [36], $G.\mathtt{rf}$ is specified to be total.

Definition 3.3 (Well-formedness)**.** An execution G is *well-formed* iff 1) $G.\mathtt{sbrf}$ is irreflexive; 2) $[G.\mathsf{B}]; G.\mathtt{sbrf}=\emptyset$; and 3) $\forall x \in \mathsf{Loc}.\, G.\mathsf{R}_x \subseteq dom(G.\mathtt{rf}) \vee G.\mathsf{W}_x \subseteq rng(G.\mathtt{rf})$. A memory model m is *well-formed* iff for all G, if $\mathsf{cons_m}(G)$ holds, then G is well-formed, and $\forall E \in \mathsf{perm}(G.\mathsf{E}).\mathsf{cons_m}(\langle E, G.\mathtt{rf}\rangle)$.

The prefix-closedness condition (**MM2**) is captured by Theorem 3.4. A consistency model m is commonly considered *prefix-closed* iff: given an execution G and a set of events $E \subseteq G.\mathsf{E}$, if $\mathsf{cons_m}(G)$ holds and $dom(G.\mathtt{sbrf}; [E]) \subseteq E$ (i.e., E is $\mathtt{sbrf}$-closed), then restricting the graph to those events in E yields a consistent execution, i.e., $\mathsf{cons_m}(G|_E)$. 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 $E = \{l_2, \mathsf{init}\}$; 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 evens $B \subseteq E$ such that the graph restricted to $E \setminus B$ is consistent: $\mathsf{cons_m}(G|_{E\setminus B})$. For instance, in the example above we can pick $B = \{l_2\}$. Note that for well-known memory models such as SC [29], TSO [34] and RC11 [24], 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, $E \subseteq G.\mathsf{E}$, if $dom(\mathtt{sbrf}; [E]) \subseteq E$ and $\mathsf{cons_m}(G)$, then there exists $B \subseteq G.\mathsf{B}$ such that $\mathsf{cons_m}(G|_{E\setminus B})$.

Memory model extensibility (**MM3**) is captured in Theorem 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.

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

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

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

- if $\langle u, r\rangle \in G.\mathtt{rf}$ and $rng([r]; G.\mathtt{sb})=\emptyset$, then there exists $w \in G.\mathsf{E} \setminus \{u\}$ such that $\mathsf{cons}(G.\mathtt{rf}[r \mapsto w])$; and
- if $\langle w, u\rangle, \langle u, u'\rangle \in G.\mathtt{rf}$ and $rng([u']; G.\mathtt{sbrf}) = \emptyset$, then $\mathsf{cons}(G|_{G.\mathsf{E}\setminus\{u\}}.\mathtt{rf}[u' \mapsto w])$.

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

From Programs to Executions Given a concurrent program, we use the same technique as [22] to pre-process it to a program of the form $P =\|_{i \in \mathsf{Tid}} P_i$, 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., [39].

Mutex Library We formulate the notion of mutex library executions and their consistency predicate in Theorem 3.6 below. For each mutex at location $l \in \mathsf{Loc}$, 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=$\langle E, rf\rangle$, we define the *mutex consistency predicate* such that it holds on G if: 1) G is well-formed (see Theorem 3.3); 2) E comprises mutex events; 3) rf is injective; and 4) rf maps lock events on to unlock events.

Intuitively, rf describes the order of mutex acquisition. For each lock event b with $\langle a, b\rangle \in rf$, 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 (3) in Theorem 3.3).

Definition 3.6. The *mutex event set* on l is $\mathsf{MX}_l \triangleq L_l \uplus U_l$ with $L_l \triangleq \{e \mid \mathtt{lab}(e)=lock(l)\}$, $U_l \triangleq \{e \mid \mathtt{lab}(e)=unlock(l)\}$.

Execution G is *mutex-consistent*, written $\mathsf{cons_{mx}}(G)$, iff: 1) G is well-formed; 2) $G.\mathsf{E}=\bigcup_{l\in\mathsf{Loc}} \mathsf{MX}_l$; 3) $G.\mathtt{rf}$ is injective; and 4) $G.\mathtt{rf}= \cup_{l\in\mathsf{Loc}}\, rf_l$ for some $rf_l \subseteq U_l \times L_l$.

It is straightforward to show that $\mathsf{cons_{mx}}(.)$ is well-formed, prefix-closed and extensible.

4 GenMC: The Generic Model Checker

In this section, we present a version of our model checking algorithm, GenMC, that does not record 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 for a version of GenMC that also tracks 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 $\langle G, T, U, S\rangle$, 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 ✓-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 of the form $\langle w, G_w\rangle$, where w denotes the revisiting write r may read from. Recall from §2.3 that upon revisiting, we restrict the graph by removing the events added to the graph after r, whilst retaining the events that lead to w in G.sbrf order; i.e., the G.sbrf prefix of w. In general, some of the events in the G.sbrf prefix of w may have been added to the graph *before* r, whilst others may have been added *after* r. As such, when removing the events added after r, we may inadvertently remove these latter events; we thus record these events in S as G_w.

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, $\mathsf{next}_P(G)$ returns an available event (Theorem 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 leftmost 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=∅, and empty maps U=S=∅ (Line 2). It then generates the executions of P

Algorithm 1 Main exploration algorithm

```
1:  procedure Verify($P$)
2:      $\langle G, T, U, S\rangle \leftarrow \langle G_0, \emptyset, \emptyset, \emptyset\rangle$
3:      VisitOne($P, G, T, U, S$)
4:      while $\langle r, w, G_w\rangle \leftarrow$ RemoveMax($S$) do
5:          $\langle E_1, r, E_2\rangle \leftarrow$ split($G$.E, $r$)
6:          $G \leftarrow \langle E_1 +\!\!+ [r] +\!\!+ G_w.\mathsf{E}, G.\mathsf{rf}|_{E_1} \cup G_w.\mathsf{rf}\rangle$
7:          $G.\mathsf{rf}[r] \leftarrow w$
8:          $T \leftarrow T \cap (E_1 +\!\!+ [r])$
9:          $U \leftarrow U \setminus \{U[r'] \mid r' \in E_2\}$
10:         if $w \neq \bot$ then CalcRevisits($G, T, U, S, r$)
11:         VisitOne($P, G, T, U, S$)
```

one at a time. This is done by calling VisitOne(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(G, T, U, S) returns the full execution generated, remaining executions are generated by exploring the options in the work list S Lines 4-11.

To do this, an option $\langle w, G_w\rangle$ 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. To see why we pick the maximal read, consider r_1, r_2, both with entries in S, such that r_1 is added to G before r_2. Let us assume that we next pick from S a revisit option for the non-maximal r_1. Recall that upon revisiting, all events added to the graph after the read are removed; we thus remove all events added after r_1, including r_2. As such, when we eventually pick from S a revisit option for r_2, the graph may no longer contain r_2, and thus we cannot perform this revisit! By always picking the maximal read, we therefore ensure that all reads in the domain of S remain in the graph at all times.

We next restrict the current graph G as described in §2.3: we split G at r (Line 5) and remove from G all events added after r (E_2), and add the events of G_w (namely the sbrf prefix of w added to the graph after r) on Line 6. Analogously, we restrict G.rf to the remaining events (E_1), add the rf edges of G_w, and finally set the rf edge of r to w (Line 7).

As the events in E_2 are removed from the graph, we accordingly remove them from the revisit set T (Line 8). We additionally remove the events in G_w. Intuitively, as the events in G_w are in the sbrf prefix of w, they are responsible for the addition of w to the graph, and consequently the reason why r is revisited by w. If any read in G_w were to be revisited, this would "undo" the revisit of r.

Analogously, Line 9 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.

Algorithm 2 Explore one program execution

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

Algorithm 3 Calculate which reads should be revisited

```
1:  procedure CalcRevisits($G, T, U, S, a$)
2:      $p_a \leftarrow dom(G.\mathtt{sbrf}^?; [a])$
3:      for $r \in T \cap \mathsf{R}_{\mathtt{loc}(a)} \setminus p_a$ do
4:          $\langle E_1, r, E_2 \rangle \leftarrow$ split($G.\mathsf{E}, r$)
5:          $G_a \leftarrow G|_{E_2 \cap p_a}$
6:          if $a \notin \mathtt{W}$ then $a \leftarrow \bot$
7:          if $\langle a, G_a \rangle \notin U[r] \land \langle a, \varnothing \rangle \notin U[r]$ then
8:              $S[r] \leftarrow S[r] \cup \{\langle a, G_a \rangle\}$
9:              $U[r] \leftarrow U[r] \cup \{\langle a, G_a \rangle\}$
```

For instance, in the FAI/2 and LOCK/2 examples, revisiting b generated additional revisit options for a—see (FAI-$\bot$) and (LOCK-$\bot$). This is done by calling CalcRevisits on Line 10. Finally, on Line 11 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 $\mathsf{next}_P(G)$, see page 8). When $\mathsf{next}_P(G)$ 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 13.

If the new event a is a write, no additional work is required. However, if a is a read, we must calculate its incoming $\mathtt{rf}$ edge. We first calculate the set of writes W that a could read from, i.e., its forward revisit options (Line 6), choose a write w_0 for the current exploration (Line 8), set a to read from w_0 in G (Line 9), add the new read to the revisit set (Line 10), and push the remaining revisit options to S (Line 11).

Note that blocking ($\bot$) is also a possible reads-from option for a. Moreover, since blocking reads do not have $\mathtt{sbrf}$ successors, if blocking is a consistent option in the current graph G, it remains a consistent option in all extensions of G. For efficiency, we thus check its consistency on the current graph G (Line 7), and add it as an option in S, if consistent (Line 11). Moreover, as blocking may also be considered as a backward revisit option for a in the future (e.g., in LOCK-$\bot$), we also add $\bot$ to the backward map U (Line 12). Note that $\bot$ is thus a special case, in that it is the only option that may be considered in both forward and backward revisits, and the *only* forward revisit option in $U[a]$.

Observe that any given write w in W is present in the graph before a is added. That is, all events in the $\mathtt{sbrf}$ prefix of w are added to the graph before a. As such, when w eventually revisits a, none of the events in its $\mathtt{sbrf}$ prefix will be removed. The associated prefix of w in S is thus *empty* $\varnothing$. That is, forward revisits, and they *alone*, have the $\varnothing$ prefix.

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. The routine leaves G unchanged with no revisits performed, and solely pushes to U and S backward revisit options that have not yet been considered, i.e., those not in U (Lines 7-9).

We iterate through all revisitable reads on the same location that a can revisit (Line 3), excluding those reads that violate $\mathtt{sbrf}$-irreflexivity; i.e., those in the $\mathtt{sbrf}$ prefix of a calculated in p_a (Line 2). Recall that for each revisit of r, we record the events in the $\mathtt{sbrf}$ prefix of the revisiting write that are added to the graph after r. To this end, we first compute the set E_2 of events added after r (Line 4), and then restrict G to events in both E_2 and the prefix p_a (Line 5).

Recall from §2.7 that when the revisiting event under consideration is a *read* and not a write, it cannot revisit existing reads in the graph itself; nevertheless, it may cause existing reads to block. For instance, in (LOCK-$\bot$), the read event b could cause a to block and we thus added $\bot$ as a revisit option for a. This is done on Line 6 by overwriting the non-write event a into the blocking option $\bot$.

Finally, recall from §4.2 that $\bot$ may be added as a *forward* revisit option for read events. When $a = \bot$, to avoid duplication, we must thus ensure that a is not already considered as a (forward) revisit option for r. This is achieved by checking that $\langle a, \varnothing \rangle$ is not included in $U[r]$ (Line 7).

4.4 GenMC: Soundness, Completeness & Optimality

The GenMC algorithm (Algorithm 1) is *sound*, *complete* and *optimal*. Given a program P and a memory model m, soundness ensures that if GenMC generates G for P under m, then $\mathsf{cons}_{\mathsf{m}}(G)$ holds; completeness ensures that if G is an execution of P under m and $\mathsf{cons}_{\mathsf{m}}(G)$ holds, then GenMC generates G for P; and optimality ensures that the P executions generated by GenMC under m are pair-wise distinct.

This is captured in the theorem below. The soundness proof is straightforward: GenMC checks consistency after each step, dropping inconsistent executions (Line 2 of VisitOne); as such, it only outputs consistent executions. The completeness and optimality proofs are non-trivial and are given in full in the technical appendix.

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

5 Evaluation

As GenMC can be instantiated for any extensible memory model, to evaluate our approach, we implemented as a verification tool for C programs 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 [24] based on wb; and
(MO) an instantiation for full RC11 that records mo.

Naturally, the generic variant is not as fast as the RC11 ones because the latter have more optimized consistency checks; it is, however, still optimal.

To evaluate GenMC, we considered CBMC [5, 13], CDSChecker [33], Herd [6], Nidhugg [2], RCMC [22], and Tracer [4]. However, among these tools, Tracer is not available and Herd is not scalable. CBMC and CDSChecker are typically significantly slower than Nidhugg and RCMC and do not scale well (CBMC because of the SAT solver, CDSChecker because of its suboptimal partial order reduction technique; see, e.g., the evaluation in [22]). For this reason, in this section, we discuss only Nidhugg (v0.2) and RCMC.

Nidhugg [2] is a state-of-the-art stateless model checker that supports SC, TSO, PSO, and POWER. Under SC, Nidhugg can also explore an optimal number of interleavings under an observational equivalence [7]. Although this equivalence class can be exponentially coarser than mo-executions (Mazurkiewicz traces [30]), it explores exponentially *more* executions compared to our WB approach. Here, due to the limited support of Nidhugg for POWER, we only compare against Nidhugg under SC[5], TSO, PSO, and SC with observational equivalence (denoted SC^o).

RCMC [22] 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.

First (§5.1), 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 (§5.2), 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 (§5.3), we perform an extensive comparison between the WB and MO variants of GenMC on 195 test cases in total. 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.

Table 1. Lamport's fast mutex algorithm [26]

	Nidhugg		RCMC		GenMC		
	SC	*SC*^{o}	*RC11*	*WRC11*	*MO*	*WB*	LIB
lamport(2)	0.19	0.21	0.10	∞	0.08	0.09	0.09
lamport(3)	12.11	7.39	5.52	∞	9.91	1.86	0.09
lamport(4)	–	–	–	∞	–	–	0.09

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 clang++-4.0 and LLVM 3.8.1 for RCMC and Nidhugg. Unless explicitly noted otherwise, all reported times are in seconds.

5.1 Model Checking Libraries

As a simple demonstration of the benefits of parametricity and compositional verification, we consider a C implementation of Lamport's fast mutual-exclusion algorithm [26] (see Table 1). We could have considered any correct lock implementation (e.g., the ones used in §5.2), but we chose this one because it has concurrent shared-variable writes (which is rare in non-synthetic programs), which render it an extremely racy lock implementation.

Nidhugg under TSO and PSO are excluded from this table for brevity, as Nidhugg-TSO is 40% slower than SC and Nidhugg-PSO crashes. The first observation is that RCMC does not terminate under WRC11. This is because this test case has writes that are never ordered under WRC11, which makes the threads' reads "oscillate" between the values of these writes ad infinitum. Note that this cannot happen under wb or mo. Additionally, both RCMC and GenMC outperform Nidhugg-SC (even though they explore more executions), with RCMC-RC11 being faster than GenMC-mo (see §5.2).

However, by feeding the axiomatic definition of the lock library to GenMC, and abstracting the inner working of the locks, GenMC is much faster than the other tools (shown in column LIB). For $N = 4$, for example, all other tools take more than 3 days to complete, whereas the generic variant of GenMC terminates almost instantly.

5.2 Overall Performance

We first demonstrate the importance of optimality (Table 2). Since RCMC is not optimal in the presence of RMWs, it can explore many more executions than necessary, which leads to significant runtime overhead. Indeed, consider the cinc

[5]Under SC, Nidhugg 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 explores very few redundant explorations, and is typically faster than optimal-DPOR.

Table 2. Some synthetic benchmarks

	Nidhugg		RCMC		GenMC	
	SC	*SC^o*	*RC11*	*WRC11*	*MO*	*WB*
cinc(4)	4.46	4.89	0.83	0.84	0.90	0.89
cinc(5)	626.14	711.84	129.09	127.25	108.59	104.67
Nw1r(5)	2.05	0.33	0.20	0.10	0.09	0.09
Nw1r(8)	1748.45	1.64	67.86	0.09	29.80	0.09

program, where all threads perform a series of RMW operations. While for 4 threads RCMC and GenMC require approximately the same time (even though RCMC explores 40% more executions), when the number of threads is increased to 5, RCMC explores almost double(!) the executions of GenMC, and this is reflected in the running time. GenMC and Nidhugg explore the same number of executions, although Nidhugg is significantly slower than the other tools, especially under observational equivalence.

We next compare the equivalence partitioning of Nidhugg and GenMC (Table 2). For Nw1r, when there are $N+1$ writers and 1 reader of a shared variable, Nidhugg-SC, RCMC-RC11, and GenMC-MO explore 5040 executions for 5 threads ($(N+2)!$). Nidhugg-SC^o explores 193, and GenMC-WB only 7. When thread number is increased to 8, Nidhugg-SC, RCMC-RC11, and GenMC-MO explore 10! executions, Nidhugg-SC^o 2305, and GenMC-MO only 10. RCMC-WRC11 explores the same number of executions as GenMC-WB but, as shown in §5.1, it fails to terminate on other benchmarks.

Next, we move to two sets of benchmarks extracted from real programs. Since Nidhugg-SC^o 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 [12, 33]. We do not show the number of executions explored because all tools explore the same number of distinct executions[6], excluding possible redundant executions explored by Nidhugg (under 5%) and duplicate executions explored by RCMC. 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.

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 [22]. That said, surprisingly, for linuxrwlocks and mpmc-queue, Nidhugg is slower in TSO than in PSO, which is a regression in the current Nidhugg version. GenMC performs similarly in terms of time under WB and MO, and explores the same

[6]Nidhugg counts the number of executions that contain a failed `assume`() statement, while RCMC does not; we take this discrepancy into account.

Table 3. Data structure benchmarks from [12, 33]

	Nidhugg			RCMC		GenMC	
	SC	*TSO*	*PSO*	*RC11*	*WRC11*	*MO*	*WB*
barrier(2)	0.27	0.27	0.27	0.09	0.08	0.11	0.11
barrier(3)	2.08	2.32	2.43	0.33	0.32	0.31	0.34
ms-queue(2)	0.72	0.77	0.91	0.19	0.20	0.20	0.13
ms-queue(3)	22.91	24.95	31.38	3.14	3.07	2.28	2.47
chase-lev	2.37	2.62	13.68	0.36	0.40	0.26	0.26
linuxrwlocks(2)	0.48	0.48	0.54	0.14	0.14	0.13	0.18
linuxrwlocks(3)	43.11	140.01	59.95	4.88	4.93	5.24	15.47
mpmc-queue(2)	0.27	0.27	0.30	0.12	0.11	0.12	0.11
mpmc-queue(3)	225.29	594.62	312.35	65.94	66.04	80.22	89.17

`barrier(N)`: A barrier implemented as a global flag with N threads that spinning and continuing only when all threads have reached the barrier.
`ms-queue(N)`: The Michael-Scott queue with N threads, each enqueing and (possibly) dequeuing an item.
`chase-lev`: An implementation of the Chase-Lev deque with four threads that concurrently operate (push, pop, steal) on the deque.
`linuxrwlocks(N)`: A reader-writer lock ported from the Linux kernel. N threads read and/or write a shared variable while holding the lock.
`mpmc-queue(N)`: A multiple-producer, multiple-consumer queue with N threads that enqueue and (possibly) dequeue.

number of executions. For linuxrwlocks, however, the WB verification requires much more time than MO. We believe this to be due to the way wb is calculated in our implementation, in cases where there are long chains of RMW events.

Finally, GenMC and RCMC seem to perform similarly in most cases, in spite of RCMC's inoptimality. Surprisingly, in the case of mpmc-queue, RCMC outperforms GenMC although it explores about 45% more executions. 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. However, this also depends on the nature of the benchmark, and the backward revisits that take place. For example, in the case of ms-queue (or N1wr from Table 2), RCMC explores no duplicates, but is still slower than GenMC.

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 `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 a few duplicate executions for RCMC in barrier and seqlock-atomic. 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 thus Nidhugg-PSO, GenMC-MO, and RCMC-MO explore more executions than Nidhugg-SC and Nidhugg-TSO (approximately 15% more). Nonetheless, GenMC and

Table 4. Benchmarks extracted from the Linux-kernel

	Nidhugg			RCMC		GenMC	
	SC	*TSO*	*PSO*	*RC11*	*WRC11*	*MO*	*WB*
mcs_spinlock(2)	0.23	0.25	0.26	0.08	0.07	0.11	0.09
mcs_spinlock(3)	5.61	7.10	12.00	0.96	0.84	0.89	0.78
mcs_spinlock(4)	1.28h	59.97h	X	0.15h	0.14h	0.85h	0.52h
qspinlock(2)	0.26	0.26	0.26	0.05	0.07	0.11	0.09
qspinlock(3)	16.81	19.88	23.11	2.43	2.50	2.24	2.61
seqlock(2)	0.22	0.24	0.26	0.11	0.10	0.11	0.11
seqlock(3)	3.03	3.22	10.49	0.56	0.59	0.74	0.75

`mcs_spinlock(N)`: An implementation of an MCS lock [31].
`qspinlock(N)`: Queued spinlocks (1.2 KLOC) are the basic spinlock implementation currently used in the Linux kernel, rendering the code in this test case heavily deployed in production. The implementation is non-trivial, as it is based on an MCS lock, but tweaked in order to further reduce cache contention and the spinlock size (it fits in only 32 bits).
`seqlock(N)`: Sequenced locks [9] (1.0 KLOC)

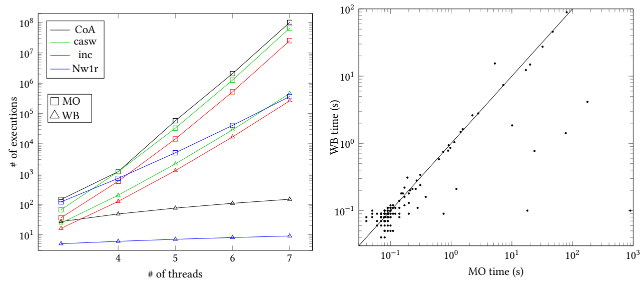


Figure 5. Comparison between GenMC-WB and MO

RCMC outperform Nidhugg by a large factor. Second, when $N = 4$, Nidhugg-TSO takes surprisingly long to finish and Nidhugg-PSO crashes; these are both due to a regression in the current version of Nidhugg. Third, GenMC-WB and RCMC-WRC11 explore *fewer* executions than GenMC-MO and RCMC-RC11, and shows the benefit of not recording mo in terms of verification time. Last, GenMC is slower than RCMC on this particular benchmark; as discussed, this is due to the number of backward revisits involved.

5.3 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 mo. As we show, however, (a) it can lead to exploring *exponentially fewer* executions than recording 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.

To see (b), consider Fig. 5 (right). This scatter diagram contains all 195 benchmarks that we used (including those of §5.2). With the only noticeable exception being linuxrwlocks (see §5.2), we can see that GenMC-WB is never much slower than GenMC-MO. On the other hand, there are many test cases where GenMC-WB is much faster than GenMC-MO.

The speedup is due to the presence of unordered concurrent writes in the program. Kokologiannakis et al. [22] 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 mcs_spinlock) where there is a difference (although not exponential) in the number of explored executions between GenMC-WB and GenMC-MO, that 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 wb and mo executions can be beneficial to spot such errors.

6 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 three basic assumptions about the underlying memory model: sbrf-acyclicity, extensibility, and prefix-closedness. 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 [35] (that do not satisfy sbrf-acyclicity) and library specifications such as queues [36] (that are not prefix-closed).

Amongst the verification tools handling weak memory models (MMs), the only properly MM-parametric tool is Herd [6]. Unlike GenMC, it does not require MMs to be extensible and thus accepts a wider range of models. Nevertheless, it follows the naive approach of enumerating *all* possible executions and filtering them according to the user-supplied consistency predicate, and thus is not scalable [22].

As discussed in §2, several tools based on *stateless model checking* [17, 18, 32] combined with (dynamic) partial order reduction (DPOR) techniques [1, 15] have targeted specific memory models [2–4, 14, 22, 33, 40]. 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 mo-executions) are: Tracer [4] for the release-acquire fragment of C11; DC-DPOR [11] for SC; and RCMC-WRC11 [22], which cannot enforce coherence and is optimal only in the absence of RMW and SC accesses. GenMC follows the general design of RCMC, but uses a revisit procedure akin to that of Tracer. As a result, our soundness proof (unlike that of RCMC) does not require "prefix-determinacy" [22, Lemma 3.9], which does not hold for the entire RC11 model: the weaker "prefix-closedness" suffices.

Other tools, such as CBMC [13], encode all executions of a program together with the memory model in a very large 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 SAT/SMT component, which can be slow and highly unpredictable. As a result, CBMC tends to be significantly slower than Nidhugg on relevant benchmarks [23, 27].

Another approach is *maximal causality reduction* (MCR) [19, 20], which introduces an even coarser equivalence partitioning than `sbrf`, based on *values* and not the places reads read-from. This approach fundamentally assumes multi-copy-atomicity, and thus cannot work for models such as RC11 [24]. On the other hand, it does work well for SC, TSO, and PSO.

Finally, unfolding-based techniques [21, 37] 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.

References

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

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

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

[4] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, and Tuan Phong Ngo. 2018. Optimal Stateless Model Checking Under the Release-acquire Semantics. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 135 (Oct. 2018), 29 pages. https://doi.org/10.1145/3276505

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

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

[7] Stavros Aronis, Bengt Jonsson, Magnus Lång, and Konstantinos Sagonas. 2018. Optimal Dynamic Partial Order Reduction with Observers. In *Tools and Algorithms for the Construction and Analysis of Systems*, Dirk Beyer and Marieke Huisman (Eds.). Springer International Publishing, Cham, 229–248.

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

[9] Hans-Juergen Boehm. 2012. Can seqlocks get along with programming language memory models?. In *MSPC 2012*. ACM, 12–20.

[10] Hans-J. Boehm and Brian Demsky. 2014. Outlawing Ghosts: Avoiding Out-of-thin-air Results. In *MSPC 2014*. ACM, New York, NY, USA, Article 7, 6 pages. https://doi.org/10.1145/2618128.2618134

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

[12] David Chase and Yossi Lev. 2005. Dynamic circular work-stealing deque. In *SPAA*.

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

[14] Brian Demsky and Patrick Lam. 2015. SATCheck: SAT-directed Stateless Model Checking for SC and TSO. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015)*. ACM, New York, NY, USA, 20–36. https://doi.org/10.1145/2814270.2814297

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

[16] Phillip B. Gibbons and Ephraim Korach. 1997. Testing Shared Memories. *SIAM J. Comput.* 26, 4 (Aug. 1997), 1208–1244. https://doi.org/10.1137/S0097539794279614

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

[18] Patrice Godefroid. 2005. Software Model Checking: The VeriSoft Approach. *Formal Methods in System Design* 26, 2 (March 2005), 77–101. https://doi.org/10.1007/s10703-005-1489-x

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

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

[21] Kari Kähkönen, Olli Saarikivi, and Keijo Heljanko. 2015. Unfolding Based Automated Testing of Multithreaded Programs. *Autom. Softw. Eng.* 22, 4 (Dec. 2015), 475–515. https://doi.org/10.1007/s10515-014-0150-6

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

[23] Michalis Kokologiannakis and Konstantinos Sagonas. 2017. Stateless Model Checking of the Linux Kernel's Hierarchical Read-copy-update (Tree RCU). In *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software (SPIN 2017)*. ACM, New York, NY, USA, 172–181. https://doi.org/10.1145/3092282.3092287

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

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

[26] Leslie Lamport. 1987. A Fast Mutual Exclusion Algorithm. *ACM Trans. Comput. Syst.* 5, 1 (Jan. 1987), 1–11. https://doi.org/10.1145/7351.7352

[27] L. Liang, P. E. McKenney, D. Kroening, and T. Melham. 2018. Verification of tree-based hierarchical read-copy update in the Linux kernel. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*. 61–66. https://doi.org/10.23919/DATE.2018.8341980

[28] Jeremy Manson, William Pugh, and Sarita V. Adve. 2005. The Java memory model. In *POPL*. 378–391.

[29] Daniel Marino, Abhayendra Singh, Todd D. Millstein, Madanlal Musuvathi, and Satish Narayanasamy. 2011. A case for an SC-preserving compiler. In *PLDI*. 199–210.

[30] Antoni Mazurkiewicz. 1987. Trace Theory. In *Petri Nets: Applications and Relationships to Other Models of Concurrency (LNCS)*, Vol. 255. Springer, Berlin, Heidelberg, 279–324. https://doi.org/10.1007/3-540-17906-2_30

[31] John M. Mellor-Crummey and Michael L. Scott. 1991. Algorithms for Scalable Synchronization on Shared-memory Multiprocessors. *ACM Trans. Comput. Syst.* 9, 1 (Feb. 1991), 21–65. https://doi.org/10.1145/103727.103729

[32] Mandanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerald Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. 2008. Finding and Reproducing Heisenbugs in Concurrent Programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, Berkeley, CA, USA, 267–280.

[33] Brian Norris and Brian Demsky. 2013. CDSchecker: checking concurrent data structures written with C/C++ atomics. In *OOPSLA 2013*. ACM, New York, NY, USA, 131–150.

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

[35] Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. 2018. Simplifying ARM concurrency: Multicopy-atomic axiomatic and operational models for ARMv8. *Proc. ACM Program. Lang.* 2, POPL (2018), 19:1–19:29. https://doi.org/10.1145/3158107

[36] Azalea Raad, Marko Doko, Lovro Rožić, Ori Lahav, and Viktor Vafeiadis. 2019. On Library Correctness under Weak Memory Consistency: Specifying and Verifying Concurrent Libraries under Declarative Consistency Models. *Proc. ACM Program. Lang.* 3, POPL (2019), 29.

[37] César Rodríguez, Marcelo Sousa, Subodh Sharma, and Daniel Kroening. 2015. Unfolding-based Partial Order Reduction. In *26th International Conference on Concurrency Theory, CONCUR 2015 (LIPIcs)*, Vol. 42. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 456–469. https://doi.org/10.4230/LIPIcs.CONCUR.2015.456

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

[39] Viktor Vafeiadis and Chinmay Narayan. 2013. Relaxed Separation Logic: A program logic for C11 concurrency. In *OOPSLA 2013*. ACM, New York, NY, USA, 867–884. https://doi.org/10.1145/2509136.2509532

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